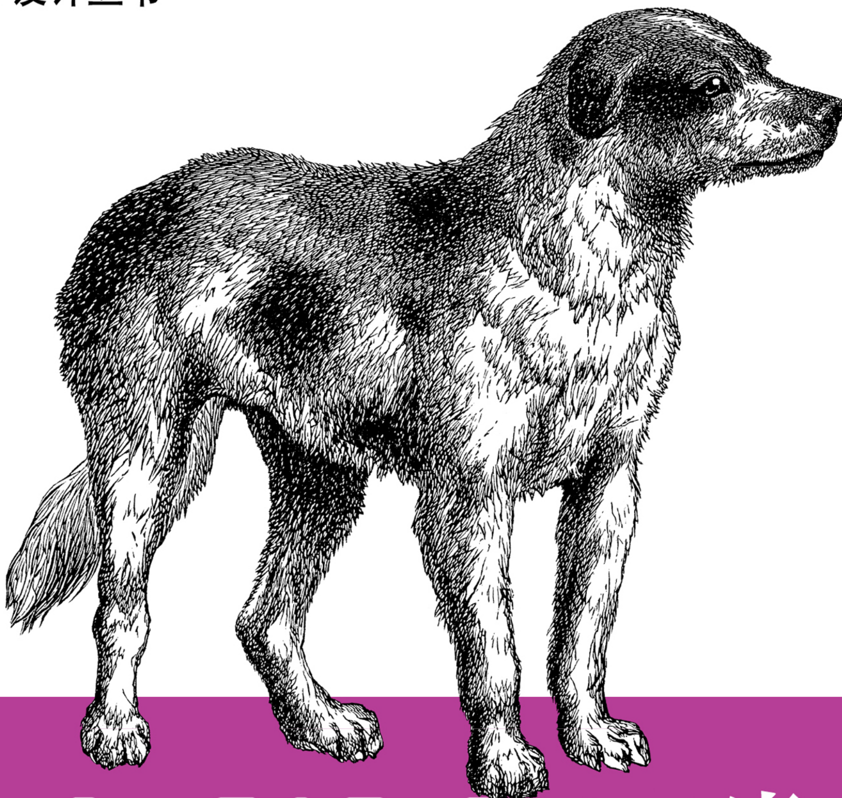
O'REILLY®

TURING 图灵程序设计丛书



Flask Web开发

基于Python的Web应用开发实战

Flask Web Development: Developing Web Applications with Python

[美] Miguel Grinberg 著

安道 译

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍



安道

人子人夫人父，机械工程师，翻译爱好者，偶尔写代码。

TURING 图灵程序设计丛书

Flask Web开发：基于Python的Web应用开发实战

Flask Web Development: Developing Web Applications with Python

[美] Miguel Grinberg 著
安道 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北京

图书在版编目（CIP）数据

Flask Web开发 ：基于Python的Web应用开发实战 /
(美) 格林布戈 (Grinberg,M.) 著 ; 安道译. -- 北京 :
人民邮电出版社, 2015.1
（图灵程序设计丛书）
ISBN 978-7-115-37399-1

Ⅰ. ①F… Ⅱ. ①格… ②安… Ⅲ. ①软件工具—程序
设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2014)第246425号

内 容 提 要

本书共分三部分，全面介绍如何基于 Python 微框架 Flask 进行 Web 开发。第一部分是 Flask 简介，介绍使用 Flask 框架及扩展开发 Web 程序的必备基础知识；第二部分则给出一个实例，真正带领大家一步步开发完整的博客和社交应用 Flasky，从而将前述知识融会贯，付诸实践。第三部分介绍了发布应用之前必须考虑的事项，如单元测试策略、性能分析技术、Flask 程序的部署方式等。

本书适合熟悉 Python 编程，有意通过 Flask 全面掌控 Web 开发的程序员学习参考。

◆ 著　　　　[美] Miguel Grinberg
译　　　　安　道
责任编辑　李松峰　毛倩倩
执行编辑　张　庆
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：14
字数：332千字　　　　　　2015年 1 月第 1 版
印数：1 – 3 000册　　　　2015年 1 月北京第 1次印刷
著作权合同登记号　图字：01-2014-5611号

定价：59.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分　实例：社交博客程序

第三部分 成功在望

前言

和其他框架相比，Flask 之所以能脱颖而出，原因在于它让开发者做主，使其能对程序具有全面的创意控制。或许你曾听过“和框架斗争”这一说法。在大多数框架中，当你决定使用的解决方案不受框架官方支持时就会发生这种情况。你可能想使用不同的数据库引擎或者不同的用户认证方法。但是，这种偏离框架开发者设定路线的做法往往会给你带来很多麻烦。

Flask 就不一样了。你喜欢关系型数据库？很好。Flask 支持所有的关系型数据库。或许你更喜欢使用 NoSQL 数据库？没问题，Flash 也支持。想使用自己开发的数据库引擎？根本用不到数据库？依然没问题。在 Flask 中，你可以自主选择程序的组件，如果找不到合适的，还可以自己开发。就这么简单。

Flask 之所以能给用户提供这么大的自由度，关键在于其开发伊始就考虑到了扩展性。Flask 提供了一个强健的核心，其中包含每个 Web 程序都需要的基本功能，而其他功能则交给行业系统中的众多第三方扩展，当然，你也可以自行开发。

在本书中，我展示自己使用 Flask 开发 Web 程序的工作流程。我不觉得这是使用 Flask 开发程序的唯一正确方式。你应该把我的选择作为一种推荐方式，而不是真理。

大部分软件开发类图书都使用短而精的示例代码，孤立地演示所介绍技术的功能，让读者自己去思考如何使用“胶水”代码把这些不同的功能结合起来，从而开发出完整可用的程序。在本书中，我采用了完全不同的方式。我使用的示例代码都摘自同一个程序，开始时很简单，后续逐章进行扩展。最初这个程序只有几行代码，最后将变成功能完善的博客和社交网络程序。

面向的读者群

要想很好地理解本书内容，你需要具备一定的 Python 编程经验。阅读本书并不要求你了解 Flask 的相关知识，但你最好能理解 Python 中的一些概念，例如包、模块、函数、修饰器

和面向对象编程。熟悉异常处理，知道如何从栈跟踪中分析问题也对理解本书有帮助。

学习本书示例代码时，你大部分时间都要在命令行中进行操作。因此，你应该能够熟练使用自己操作系统中的命令行。

现代 Web 程序都不可避免地需要使用 HTML、CSS 和 JavaScript。本书开发的示例程序当然也用到了这些技术，但本书没有对其进行详细介绍，也没有说明应该如何使用。因此，如果你想开发完整的程序，且无法向精通客户端技术的开发者寻求帮助，那就需要对这些语言有一定程度的了解。

本书附带的程序是开源的，我把它上传到了 GitHub。虽然可以从 GitHub 上下载 ZIP 或 TAR 格式的程序源码，但我还是强烈建议你安装 Git 客户端，以便熟悉怎么使用源码版本控制系统，至少知道如何直接从仓库中克隆源码以及如何切换到程序的不同版本。接下来的“如何使用示例代码”部分会介绍几个你需要知道的命令。你或许希望在自己的项目中使用版本控制，那就把本书作为学习 Git 的一个契机吧。

最后要说明的是，本书并不是完整且详尽介绍 Flask 框架的手册。本书介绍了 Flask 的大部分功能，但你还需要配合使用 Flask 官方文档（http://flask.pocoo.org/）。

本书结构

本书分为三部分。

第一部分　Flask 简介　简要介绍如何使用 Flask 框架及其一些扩展开发 Web 程序。

- 第 1 章　说明如何安装和设置 Flask 框架；
- 第 2 章　通过一个简单的程序介绍如何使用 Flask；
- 第 3 章　介绍如何在 Flask 程序中使用模板；
- 第 4 章　介绍 Web 表单；
- 第 5 章　介绍数据库；
- 第 6 章　介绍如何实现电子邮件支持；
- 第 7 章　提供一个可供中大型程序使用的程序结构。

第二部分　实例：社交博客程序　开发 Flasky，这是我为本书开发的开源博客和社交网络程序。

- 第 8 章　实现用户认证系统；
- 第 9 章　实现用户角色和权限；
- 第 10 章　实现用户资料页；
- 第 11 章　开发博客界面；

- 第 12 章　实现关注功能；
- 第 13 章　实现博客文章的用户评论功能；
- 第 14 章　实现应用编程接口（Application Programming Interface，API）。

第三部分　成功在望　介绍与开发程序没有直接关系，但在程序发布之前要考虑的事项。

- 第 15 章　详细说明各种单元测试策略；
- 第 16 章　简要介绍性能分析技术；
- 第 17 章　说明 Flask 程序的部署方式，包含传统方式和云方式；
- 第 18 章　列出其他资源。

如何使用示例代码

本书使用的示例代码可在 GitHub 上获取[1]：https://github.com/miguelgrinberg/flasky。

这个仓库的提交历史被精心设置为与本书所介绍的功能顺序一致。使用这份代码时，我建议你从最早的提交开始，顺着本书内容的进度，向前推移查看提交列表。另外，你还可以从 GitHub 上下载每次提交代码后得到的 ZIP 或 TAR 文件。

如果你决定使用 Git 操作源码，那么首先要安装 Git 客户端（可以从 http://git-scm.com/ 下载）。下述命令就使用 Git 下载示例代码：

```
$ git clone https://github.com/miguelgrinberg/flasky.git
```

`git clone` 命令从 GitHub 上下载源码，安装到当前目录下的 flasky 文件夹中。这个文件夹中不仅有源码，还有一个包含了程序修改完整历史的 Git 仓库。

第 1 章会要求你签出程序的初始发布版本，然后在适当的时候指示你需要向前推进查看提交历史。切换提交历史的 Git 命令是 `git checkout`。下面举个例子：

```
$ git checkout 1a
```

上述命令中的 `1a` 代表一个标签（tag），是项目中某次提交历史的名字。这个仓库的标签根据本书的章节命名，因此本例中的 `1a` 表示第 1 章使用程序的初始版本。大多数章都不止使用一个标签，例如 `5a` 和 `5b` 等分别对应第 5 章中使用到的不同版本。

除了签出程序源码的不同版本，你可能还需要进行一些设置。例如，你有时需要安装额外的 Python 包，或者升级数据库。需要执行这些操作时，我会提醒你。

一般情况下，你无需修改程序的源文件，但如果修改了，Git 会阻止你签出其他历史版本，因为这会导致本地修改历史的丢失。签出其他历史版本之前，你要把文件还原到原始状

编注 1：也可注册 iTuring.cn，在本书页面免费下载。

态。最简单的方法是使用 `git reset` 命令：

```
$ git reset --hard
```

这个命令会损坏本地修改，所以执行此命令前你需要保存所有不想丢失的改动。

你可能经常需要从 GitHub 上下载修正和改进后的源码用于更新本地仓库。完成这个操作的命令如下所示：

```
$ git fetch --all
$ git fetch --tags
$ git reset --hard origin/master
```

`git fetch` 命令用于利用 GitHub 上的远程仓库更新本地仓库的提交历史和标签，但不会改动真正的源文件，随后执行的 `git reset` 命令才是用于更新文件的操作。再次提醒，执行 `git reset` 命令后，本地修改会丢失。

另一个有用的操作是查看程序两个版本之间的区别，以便了解改动详情。在命令行中，你可以使用 `git diff` 命令进行查看。例如，执行下述命令可以查看 `2a` 和 `2b` 两个修订版本之间的区别：

```
$ git diff  2a 2b
```

这个命令以补丁（patch）的形式显示区别，如果你以前没有用过补丁文件，可能会觉得这种查看变动的方式不直观。你可能发现，GitHub 网站中显示的图形化对比更容易让人理解。例如，在 GitHub 中查看 `2a` 和 `2b` 两个历史版本的区别，可以访问 https://github.com/miguelgrinberg/flasky/compare/2a...2b。

使用代码示例

本书的目的是帮助你完成工作。一般来说，你可以在自己的程序或者文档中使用本书附带的示例代码。你无需联系我们获得使用许可，除非你要复制大量的代码。例如，使用本书中的多个代码片段编写程序就无需获得许可。但以 CD-ROM 的形式销售或者分发 O'Reilly 书中的示例代码则需要获得许可。回答问题时援引本书内容以及书中示例代码，无需获得许可。在你自己的项目文档中使用本书大量的示例代码时，则需要获得许可。

我们不强制要求署名，但如果你这么做，我们深表感激。署名一般包括书名、作者、出版社和国际标准图书编号。例如：*Flask Web Development* by Miguel Grinberg (O'Reilly). Copyright 2014 Miguel Grinberg, 978-1-449-3726-2。

如果你觉得自身情况不在合理使用或上述允许的范围内，请通过邮件和我们联系，地址是 permissions@oreilly.com。

排版约定

本书使用了下述排版约定。

楷体
标示新术语。

等宽字体（`constant width`）
表示程序代码，也表示正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

加粗等宽字体（**`constant width`**）
命令或是其他应该由用户输入的内容。

斜体等宽字体（*`constant width`*）
表示需要使用用户的输入值代替的文本，或者由上下文决定的值。

这个图标表示提示或建议。

这个图标表示一般性说明。

这个图标表示警告或提醒。

Safari® Books Online

Safari Books Online（http://my.safaribooksonline.com/?portal=oreilly）是应需而变的数字图书馆，它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品（http://www.safaribooksonline.com/content）。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展

调研、解决问题、学习和认证培训的第一手资料。

Safari 在线图书馆为组织（http://www.safaribooksonline.com/enterprise）、政府部门（http://www.safaribooksonline.com/government）和个人（http://www.safaribooksonline.com/）提供了不同的产品组合和灵活的定价策略。订阅者可以在一个开放搜索的全文数据库中访问上千种图书、培训视频和正式出版之前的书稿。这些内容由以下出版社提供：O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology，等等（http://www.safaribooksonline.com/publishers）。若想了解关于 Safari Books Online 的更多信息，请访问我们的网站（http://www.safaribooksonline.com）。

联系我们

请把对本书的意见和疑问发送给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920031116.do。

如果你对本书有一些建议或技术上的疑问，请发送电子邮件至 bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

我一个人是无法完成这本书的。家人、同事、老友，以及写书过程中认识的新朋友都给了我很大的帮助。

我要感谢 Brendan Kohler，他对本书做了详尽的技术审校，并为第 14 章的成型提供了宝贵建议。我还要感谢 David Baumgold、Todd Brunhoff、Cecil Rock 和 Matthew Hugues，他们在本书撰写的不同阶段审阅了书稿，并对涵盖内容和组织方式给予了建设性建议。

本书示例代码的编写花费了我的大量精力。我很感激 Daniel Hofmann 的帮助，他对这个程序做了完整的代码审查，并指出了很多可改进之处。还要感谢我十几岁的儿子 Dylan Grinberg，他暂时克服了 Minecraft 这款游戏的强大吸引力，用几周时间帮助我在不同平台上测试这些代码。

O'Reilly 有个极好的项目名为 Early Release（提早发布），可以让迫不及待的读者在图书撰写过程中就进行阅读。很多抢先阅读的读者并不局限于阅读本书，还加入了讨论行列，讨论他们使用本书的体验，这为本书的改进做出了极大贡献。在这些读者中，我要特别感谢 Sundeep Gupta、Dan Caron、Brian Wisti 和 Cody Scott 对本书所做的贡献。

O'Reilly Media 的工作人员始终陪伴着我。首先我要特别感谢本书的编辑 Meghan Blanchette，她从我们见面的第一天起，就给予我无尽的支持、建议和协助。她把我写作第一本书的过程变成了美好的回忆。

最后，请让我对 Flask 社区表示由衷的感谢。

第一部分

Flask简介

第 1 章 安装

在大多数标准中，Flask（http://flask.pocoo.org/）都算是小型框架，小到可以称为“微框架”。Flask 非常小，因此你一旦能够熟练使用它，很可能就能读懂它所有的源码。

但是，小并不意味着它比其他框架的功能少。Flask 自开发伊始就被设计为可扩展的框架，它具有一个包含基本服务的强健核心，其他功能则可通过扩展实现。你可以自己挑选所需的扩展包，组成一个没有附加功能的精益组合，从而完全精确满足自身需求。

Flask 有两个主要依赖：路由、调试和 Web 服务器网关接口（Web Server Gateway Interface，WSGI）子系统由 Werkzeug（http://werkzeug.pocoo.org/）提供；模板系统由 Jinja2（http://jinja.pocoo.org/）提供。Werkzeug 和 Jinjia2 都是由 Flask 的核心开发者开发而成。

Flask 并不原生支持数据库访问、Web 表单验证和用户认证等高级功能。这些功能以及其他大多数 Web 程序中需要的核心服务都以扩展的形式实现，然后再与核心包集成。开发者可以任意挑选符合项目需求的扩展，甚至可以自行开发。这和大型框架的做法相反，大型框架往往已经替你做出了大多数决定，难以（有时甚至不允许）使用替代方案。

本章介绍如何安装 Flask。在这一学习过程中，你只需要一台安装了 Python 的电脑。

> 本书中的代码示例已经通过 Python 2.7 和 Python 3.3 的测试，所以我们强烈建议大家选用这两个版本。

1.1　使用虚拟环境

安装 Flask 最便捷的方式是使用虚拟环境。虚拟环境是 Python 解释器的一个私有副本，在这个环境中你可以安装私有包，而且不会影响系统中安装的全局 Python 解释器。

虚拟环境非常有用，可以在系统的 Python 解释器中避免包的混乱和版本的冲突。为每个程序单独创建虚拟环境可以保证程序只能访问虚拟环境中的包，从而保持全局解释器的干净整洁，使其只作为创建（更多）虚拟环境的源。使用虚拟环境还有个好处，那就是不需要管理员权限。

虚拟环境使用第三方实用工具 `virtualenv` 创建。输入以下命令可以检查系统是否安装了 `virtualenv`：

```
$ virtualenv --version
```

如果结果显示错误，你就需要安装这个工具。

> Python 3.3 通过 `venv` 模块原生支持虚拟环境，命令为 `pyvenv`。`pyvenv` 可以替代 `virtualenv`。不过要注意，在 Python 3.3 中使用 `pyvenv` 命令创建的虚拟环境不包含 `pip`，你需要进行手动安装。Python 3.4 改进了这一缺陷，`pyvenv` 完全可以代替 `virtualenv`。

大多数 Linux 发行版都提供了 `virtualenv` 包。例如，Ubuntu 用户可以使用下述命令安装它：

```
$ sudo apt-get install python-virtualenv
```

如果你的电脑是 Mac OS X 系统，就可以使用 `easy_install` 安装 `virtualenv`：

```
$ sudo easy_install virtualenv
```

如果你使用微软的 Windows 系统或其他没有官方 `virtualenv` 包的操作系统，那么安装过程要稍微复杂一点。

在浏览器中输入网址 https://bitbucket.org/pypa/setuptools，回车后会进入 `setuptools` 安装程序的主页。在这个页面中找到下载安装脚本的链接，脚本名为 ez_setup.py。把这个文件保存到电脑的一个临时文件夹中，然后在这个文件夹中执行以下命令：

```
$ python ez_setup.py
$ easy_install virtualenv
```

上述命令必须以具有管理员权限的用户身份执行。在微软 Windows 系统中，请使用“以管理员身份运行”选项打开命令行窗口；在基于 Unix 的系统中，要在上面两个命令前加上 `sudo`，或者以根用户身份执行。一旦安装完毕，`virtualenv` 实用工具就可以从常规账户中调用。

现在你要新建一个文件夹，用来保存示例代码（示例代码可从 GitHub 库中获取）。我们在前言的“如何使用示例代码”一节中说过，获取示例代码最简便的方式是使用 Git 客户端直接从 GitHub 下载。下述命令从 GitHub 下载示例代码，并把程序文件夹切换到“1a”版本，即程序的初始版本：

```
$ git clone https://github.com/miguelgrinberg/flasky.git
$ cd flasky
$ git checkout 1a
```

下一步是使用 `virtualenv` 命令在 flasky 文件夹中创建 Python 虚拟环境。这个命令只有一个必需的参数，即虚拟环境的名字。创建虚拟环境后，当前文件夹中会出现一个子文件夹，名字就是上述命令中指定的参数，与虚拟环境相关的文件都保存在这个子文件夹中。按照惯例，一般虚拟环境会被命名为 `venv`：

```
$ virtualenv venv
New python executable in venv/bin/python2.7
Also creating executable in venv/bin/python
Installing setuptools............done.
Installing pip...............done.
```

现在，flasky 文件夹中就有了一个名为 venv 的子文件夹，它保存一个全新的虚拟环境，其中有一个私有的 Python 解释器。在使用这个虚拟环境之前，你需要先将其“激活”。如果你使用 bash 命令行（Linux 和 Mac OS X 用户），可以通过下面的命令激活这个虚拟环境：

```
$ source venv/bin/activate
```

如果使用微软 Windows 系统，激活命令是：

```
$ venv\Scripts\activate
```

虚拟环境被激活后，其中 Python 解释器的路径就被添加进 `PATH` 中，但这种改变不是永久性的，它只会影响当前的命令行会话。为了提醒你已经激活了虚拟环境，激活虚拟环境的命令会修改命令行提示符，加入环境名：

```
(venv) $
```

当虚拟环境中的工作完成后，如果你想回到全局 Python 解释器中，可以在命令行提示符下输入 `deactivate`。

1.2 使用pip安装Python包

大多数 Python 包都使用 pip 实用工具安装，使用 virtualenv 创建虚拟环境时会自动安装 pip。激活虚拟环境后，pip 所在的路径会被添加进 PATH。

> 如果你在 Python 3.3 中使用 pyvenv 创建虚拟环境，那就需要手动安装 pip。安装方法参见 pip 的网站（https://pip.pypa.io/en/latest/installing.html）。在 Python 3.4 中，pyvenv 会自动安装 pip。

执行下述命令可在虚拟环境中安装 Flask：

```
(venv) $ pip install flask
```

执行上述命令，你就在虚拟环境中安装 Flask 及其依赖了。要想验证 Flask 是否正确安装，你可以启动 Python 解释器，尝试导入 Flask：

```
(venv) $ python
>>> import flask
>>>
```

如果没有看到错误提醒，那恭喜你——你已经可以开始学习第 2 章的内容，了解如何开发第一个 Web 程序了。

第 2 章

程序的基本结构

本章将带你了解 Flask 程序各部分的作用，编写并运行第一个 Flask Web 程序。

2.1　初始化

所有 Flask 程序都必须创建一个*程序实例*。Web 服务器使用一种名为 Web 服务器网关接口（Web Server Gateway Interface，WSGI）的协议，把接收自客户端的所有请求都转交给这个对象处理。程序实例是 `Flask` 类的对象，经常使用下述代码创建：

```
from flask import Flask
app = Flask(__name__)
```

`Flask` 类的构造函数只有一个必须指定的参数，即程序主模块或包的名字。在大多数程序中，Python 的 `__name__` 变量就是所需的值。

> 将构造函数的 `name` 参数传给 Flask 程序，这一点可能会让 Flask 开发新手心生迷惑。Flask 用这个参数决定程序的根目录，以便稍后能够找到相对于程序根目录的资源文件位置。

后文会介绍更复杂的程序初始化方式，对于简单的程序来说，上面的代码足够了。

2.2　路由和视图函数

客户端（例如 Web 浏览器）把请求发送给 Web 服务器，Web 服务器再把请求发送给 Flask

程序实例。程序实例需要知道对每个 URL 请求运行哪些代码，所以保存了一个 URL 到 Python 函数的映射关系。处理 URL 和函数之间关系的程序称为路由。

在 Flask 程序中定义路由的最简便方式，是使用程序实例提供的 `app.route` 修饰器，把修饰的函数注册为路由。下面的例子说明了如何使用这个修饰器声明路由：

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

> 修饰器是 Python 语言的标准特性，可以使用不同的方式修改函数的行为。惯常用法是使用修饰器把函数注册为事件的处理程序。

前例把 `index()` 函数注册为程序根地址的处理程序。如果部署程序的服务器域名为 www.example.com，在浏览器中访问 http://www.example.com 后，会触发服务器执行 `index()` 函数。这个函数的返回值称为响应，是客户端接收到的内容。如果客户端是 Web 浏览器，响应就是显示给用户查看的文档。

像 `index()` 这样的函数称为视图函数（view function）。视图函数返回的响应可以是包含 HTML 的简单字符串，也可以是复杂的表单，后文会介绍。

> 在 Python 代码中嵌入响应字符串会导致代码难以维护，此处这么做只是为了介绍响应的概念。你将在第 3 章了解生成响应的正确方法。

如果你仔细观察日常所用服务的某些 URL 格式，会发现很多地址中都包含可变部分。例如，你的 Facebook 资料页面的地址是 http://www.facebook.com/<*your-name*>，用户名（*your-name*）是地址的一部分。Flask 支持这种形式的 URL，只需在 `route` 修饰器中使用特殊的句法即可。下例定义的路由中就有一部分是动态名字：

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name
```

尖括号中的内容就是动态部分，任何能匹配静态部分的 URL 都会映射到这个路由上。调用视图函数时，Flask 会将动态部分作为参数传入函数。在这个视图函数中，参数用于生成针对个人的欢迎消息。

路由中的动态部分默认使用字符串，不过也可使用类型定义。例如，路由 /user/<int:id> 只会匹配动态片段 id 为整数的 URL。Flask 支持在路由中使用 int、float 和 path 类型。path 类型也是字符串，但不把斜线视作分隔符，而将其当作动态片段的一部分。

2.3　启动服务器

程序实例用 run 方法启动 Flask 集成的开发 Web 服务器：

```
if __name__ == '__main__':
    app.run(debug=True)
```

__name__=='__main__' 是 Python 的惯常用法，在这里确保直接执行这个脚本时才启动开发 Web 服务器。如果这个脚本由其他脚本引入，程序假定父级脚本会启动不同的服务器，因此不会执行 app.run()。

服务器启动后，会进入轮询，等待并处理请求。轮询会一直运行，直到程序停止，比如按 Ctrl-C 键。

有一些选项参数可被 app.run() 函数接受用于设置 Web 服务器的操作模式。在开发过程中启用调试模式会带来一些便利，比如说激活*调试器*和*重载程序*。要想启用调试模式，我们可以把 debug 参数设为 True。

> Flask 提供的 Web 服务器不适合在生产环境中使用。第 17 章会介绍生产环境 Web 服务器。

2.4　一个完整的程序

前几节介绍了 Flask Web 程序的不同组成部分，现在是时候开发一个程序了。整个 hello.py 程序脚本就是把前面介绍的三部分合并到一个文件中。程序代码如示例 2-1 所示。

示例 2-1　hello.py：一个完整的 Flask 程序

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

if __name__ == '__main__':
    app.run(debug=True)
```

如果你已经从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 2a` 签出程序的这个版本。

要想运行这个程序，请确保激活了你之前创建的虚拟环境，并在其中安装了 Flask。现在打开 Web 浏览器，在地址栏中输入 http://127.0.0.1:5000/。图 2-1 是浏览器连接到程序后的示意图。

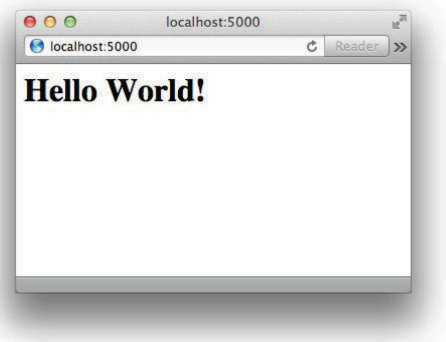


图 2-1　hello.py Flask 程序

然后使用下述命令启动程序：

```
(venv) $ python hello.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

如果你输入其他地址，程序将不知道如何处理，因此会向浏览器返回错误代码 404。访问不存在的网页时，你也会经常看到这个熟悉的错误。

示例 2-2 是这个程序的增强版，添加了一个动态路由。访问这个地址时，你会看到一则针对个人的欢迎消息。

示例 2-2　hello.py：包含动态路由的 Flask 程序

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
```

```
def user(name):
    return '<h1>Hello, %s!</h1>' % name

if __name__ == '__main__':
    app.run(debug=True)
```

> 如果你已经从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 2b` 签出程序的这个版本。

测试动态路由前，你要确保服务器正在运行中，然后访问 http://localhost:5000/user/Dave。程序会显示一个使用 `name` 动态参数生成的欢迎消息。请尝试使用不同的名字，可以看到视图函数总是使用指定的名字生成响应。图 2-2 展示了一个示例。

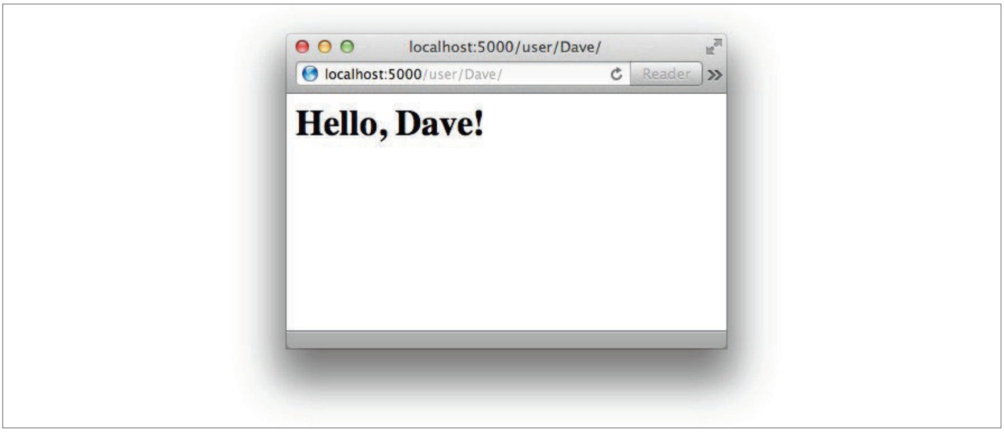


图 2-2　动态路由

2.5　请求–响应循环

现在你已经开发了一个简单的 Flask 程序，或许希望进一步了解 Flask 的工作方式。下面几节将介绍这个框架的一些设计理念。

2.5.1　程序和请求上下文

Flask 从客户端收到请求时，要让视图函数能访问一些对象，这样才能处理请求。请求对象就是一个很好的例子，它封装了客户端发送的 HTTP 请求。

要想让视图函数能够访问请求对象，一个显而易见的方式是将其作为参数传入视图函数，不过这会导致程序中的每个视图函数都增加一个参数。除了访问请求对象，如果视图函数

在处理请求时还要访问其他对象，情况会变得更糟。

为了避免大量可有可无的参数把视图函数弄得一团糟，Flask 使用上下文临时把某些对象变为全局可访问。有了上下文，就可以写出下面的视图函数：

```
from flask import request

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is %s</p>' % user_agent
```

注意在这个视图函数中我们如何把 `request` 当作全局变量使用。事实上，`request` 不可能是全局变量。试想，在多线程服务器中，多个线程同时处理不同客户端发送的不同请求时，每个线程看到的 `request` 对象必然不同。Falsk 使用上下文让特定的变量在一个线程中全局可访问，与此同时却不会干扰其他线程。

> 线程是可单独管理的最小指令集。进程经常使用多个活动线程，有时还会共享内存或文件句柄等资源。多线程 Web 服务器会创建一个线程池，再从线程池中选择一个线程用于处理接收到的请求。

在 Flask 中有两种上下文：*程序上下文*和*请求上下文*。表 2-1 列出了这两种上下文提供的变量。

表2-1　Flask上下文全局变量

变量名	上下文	说　　明
`current_app`	程序上下文	当前激活程序的程序实例
`g`	程序上下文	处理请求时用作临时存储的对象。每次请求都会重设这个变量
`request`	请求上下文	请求对象，封装了客户端发出的 HTTP 请求中的内容
`session`	请求上下文	用户会话，用于存储请求之间需要“记住”的值的词典

Flask 在分发请求之前激活（或*推送*）程序和请求上下文，请求处理完成后再将其删除。程序上下文被推送后，就可以在线程中使用 `current_app` 和 `g` 变量。类似地，请求上下文被推送后，就可以使用 `request` 和 `session` 变量。如果使用这些变量时我们没有激活程序上下文或请求上下文，就会导致错误。如果你不知道为什么这 4 个上下文变量如此有用，先别担心，后面的章节会详细说明。

下面这个 Python shell 会话演示了程序上下文的使用方法：

```
>>> from hello import app
>>> from flask import current_app
>>> current_app.name
```

```
Traceback (most recent call last):
...
RuntimeError: working outside of application context
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'hello'
>>> app_ctx.pop()
```

在这个例子中，没激活程序上下文之前就调用 `current_app.name` 会导致错误，但推送完上下文之后就可以调用了。注意，在程序实例上调用 `app.app_context()` 可获得一个程序上下文。

2.5.2 请求调度

程序收到客户端发来的请求时，要找到处理该请求的视图函数。为了完成这个任务，Flask 会在程序的 URL 映射中查找请求的 URL。URL 映射是 URL 和视图函数之间的对应关系。Flask 使用 `app.route` 修饰器或者非修饰器形式的 `app.add_url_rule()` 生成映射。

要想查看 Flask 程序中的 URL 映射是什么样子，我们可以在 Python shell 中检查为 hello.py 生成的映射。测试之前，请确保你激活了虚拟环境：

```
(venv) $ python
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
 <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

`/` 和 `/user/<name>` 路由在程序中使用 `app.route` 修饰器定义。`/static/<filename>` 路由是 Flask 添加的特殊路由，用于访问静态文件。第 3 章会详细介绍静态文件。

URL 映射中的 `HEAD`、`Options`、`GET` 是*请求方法*，由路由进行处理。Flask 为每个路由都指定了请求方法，这样不同的请求方法发送到相同的 URL 上时，会使用不同的视图函数进行处理。`HEAD` 和 `OPTIONS` 方法由 Flask 自动处理，因此可以这么说，在这个程序中，URL 映射中的 3 个路由都使用 `GET` 方法。第 4 章会介绍如何为路由指定不同的请求方法。

2.5.3 请求钩子

有时在处理请求之前或之后执行代码会很有用。例如，在请求开始时，我们可能需要创建数据库连接或者认证发起请求的用户。为了避免在每个视图函数中都使用重复的代码，Flask 提供了注册通用函数的功能，注册的函数可在请求被分发到视图函数之前或之后调用。

请求钩子使用修饰器实现。Flask 支持以下 4 种钩子。

- `before_first_request`：注册一个函数，在处理第一个请求之前运行。
- `before_request`：注册一个函数，在每次请求之前运行。
- `after_request`：注册一个函数，如果没有未处理的异常抛出，在每次请求之后运行。
- `teardown_request`：注册一个函数，即使有未处理的异常抛出，也在每次请求之后运行。

在请求钩子函数和视图函数之间共享数据一般使用上下文全局变量 `g`。例如，`before_request` 处理程序可以从数据库中加载已登录用户，并将其保存到 `g.user` 中。随后调用视图函数时，视图函数再使用 `g.user` 获取用户。

请求钩子的用法会在后续章中介绍，如果你现在不太理解，也不用担心。

2.5.4　响应

Flask 调用视图函数后，会将其返回值作为响应的内容。大多数情况下，响应就是一个简单的字符串，作为 HTML 页面回送客户端。

但 HTTP 协议需要的不仅是作为请求响应的字符串。HTTP 响应中一个很重要的部分是*状态码*，Flask 默认设为 200，这个代码表明请求已经被成功处理。

如果视图函数返回的响应需要使用不同的状态码，那么可以把数字代码作为第二个返回值，添加到响应文本之后。例如，下述视图函数返回一个 400 状态码，表示请求无效：

```
@app.route('/')
def index():
    return '<h1>Bad Request</h1>', 400
```

视图函数返回的响应还可接受第三个参数，这是一个由首部（header）组成的字典，可以添加到 HTTP 响应中。一般情况下并不需要这么做，不过你会在第 14 章看到一个例子。

如果不想返回由 1 个、2 个或 3 个值组成的元组，Flask 视图函数还可以返回 `Response` 对象。`make_response()` 函数可接受 1 个、2 个或 3 个参数（和视图函数的返回值一样），并返回一个 `Response` 对象。有时我们需要在视图函数中进行这种转换，然后在响应对象上调用各种方法，进一步设置响应。下例创建了一个响应对象，然后设置了 cookie：

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('answer', '42')
    return response
```

有一种名为*重定向*的特殊响应类型。这种响应没有页面文档，只告诉浏览器一个新地址用以加载新页面。重定向经常在 Web 表单中使用，第 4 章会进行介绍。

重定向经常使用 302 状态码表示，指向的地址由 Location 首部提供。重定向响应可以使用 3 个值形式的返回值生成，也可在 Response 对象中设定。不过，由于使用频繁，Flask 提供了 redirect() 辅助函数，用于生成这种响应：

```
from flask import redirect

@app.route('/')
def index():
    return redirect('http://www.example.com')
```

还有一种特殊的响应由 abort 函数生成，用于处理错误。在下面这个例子中，如果 URL 中动态参数 id 对应的用户不存在，就返回状态码 404：

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort(404)
    return '<h1>Hello, %s</h1>' % user.name
```

注意，abort 不会把控制权交还给调用它的函数，而是抛出异常把控制权交给 Web 服务器。

2.6 Flask扩展

Flask 被设计为可扩展形式，故而没有提供一些重要的功能，例如数据库和用户认证，所以开发者可以自由选择最适合程序的包，或者按需求自行开发。

社区成员开发了大量不同用途的*扩展*，如果这还不能满足需求，你还可使用所有 Python 标准包或代码库。为了让你知道如何把扩展整合到程序中，接下来我们将在 hello.py 中添加一个扩展，使用命令行参数增强程序的功能。

使用Flask-Script支持命令行选项

Flask 的开发 Web 服务器支持很多启动设置选项，但只能在脚本中作为参数传给 app.run() 函数。这种方式并不十分方便，传递设置选项的理想方式是使用命令行参数。

Flask-Script 是一个 Flask 扩展，为 Flask 程序添加了一个命令行解析器。Flask-Script 自带了一组常用选项，而且还支持自定义命令。

Flask-Script 扩展使用 pip 安装：

```
(venv) $ pip install flask-script
```

示例 2-3 显示了把命令行解析功能添加到 hello.py 程序中时需要修改的地方。

示例 2-3　hello.py：使用 Flask-Script

```
from flask.ext.script import Manager
manager = Manager(app)

# ...

if __name__ == '__main__':
    manager.run()
```

专为 Flask 开发的扩展都暴漏在 `flask.ext` 命名空间下。Flask-Script 输出了一个名为 `Manager` 的类，可从 `flask.ext.script` 中引入。

这个扩展的初始化方法也适用于其他很多扩展：把程序实例作为参数传给构造函数，初始化主类的实例。创建的对象可以在各个扩展中使用。在这里，服务器由 `manager.run()` 启动，启动后就能解析命令行了。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 2c` 签出程序的这个版本。

这样修改之后，程序可以使用一组基本命令行选项。现在运行 hello.py，会显示一个用法消息：

```
$ python hello.py
usage: hello.py [-h] {shell,runserver} ...

positional arguments:
  {shell,runserver}
    shell           在 Flask 应用上下文中运行 Python shell
    runserver       运行 Flask 开发服务器：app.run()

optional arguments:
  -h, --help        显示帮助信息并退出
```

`shell` 命令用于在程序的上下文中启动 Python shell 会话。你可以使用这个会话中运行维护任务或测试，还可调试异常。

顾名思义，`runserver` 命令用来启动 Web 服务器。运行 `python hello.py runserver` 将以调试模式启动 Web 服务器，但是我们还有很多选项可用：

```
(venv) $ python hello.py runserver --help
usage: hello.py runserver [-h] [-t HOST] [-p PORT] [--threaded]
                          [--processes PROCESSES] [--passthrough-errors] [-d]
                          [-r]
```

```
运行 Flask 开发服务器：app.run()

optional arguments:
  -h, --help            显示帮助信息并退出
  -t HOST, --host HOST
  -p PORT, --port PORT
  --threaded
  --processes PROCESSES
  --passthrough-errors
  -d, --no-debug
  -r, --no-reload
```

`--host` 参数是个很有用的选项，它告诉 Web 服务器在哪个网络接口上监听来自客户端的连接。默认情况下，Flask 开发 Web 服务器监听 `localhost` 上的连接，所以只接受来自服务器所在计算机发起的连接。下述命令让 Web 服务器监听公共网络接口上的连接，允许同网中的其他计算机连接服务器：

```
(venv) $ python hello.py runserver --host 0.0.0.0
 * Running on http://0.0.0.0:5000/
 * Restarting with reloader
```

现在，Web 服务器可使用 http://a.b.c.d:5000/ 网络中的任一台电脑进行访问，其中“a.b.c.d”是服务器所在计算机的外网 IP 地址。

本章介绍了请求响应的概念，不过响应的知识还有很多。对于使用模板生成响应，Flask 提供了良好支持，这是个很重要的话题，下一章我们还要专门介绍模板。

第 3 章
模板

要想开发出易于维护的程序，关键在于编写形式简洁且结构良好的代码。到目前为止，你看到的示例都太简单，无法说明这一点，但 Flask 视图函数的两个完全独立的作用却被融合在了一起，这就产生了一个问题。

视图函数的作用很明确，即生成请求的响应，如第 2 章中的示例所示。对最简单的请求来说，这就足够了，但一般而言，请求会改变程序的状态，而这种变化也会在视图函数中产生。

例如，用户在网站中注册了一个新账户。用户在表单中输入电子邮件地址和密码，然后点击提交按钮。服务器接收到包含用户输入数据的请求，然后 Flask 把请求分发到处理注册请求的视图函数。这个视图函数需要访问数据库，添加新用户，然后生成响应回送浏览器。这两个过程分别称为*业务逻辑*和*表现逻辑*。

把业务逻辑和表现逻辑混在一起会导致代码难以理解和维护。假设要为一个大型表格构建 HTML 代码，表格中的数据由数据库中读取的数据以及必要的 HTML 字符串连接在一起。把表现逻辑移到*模板*中能够提升程序的可维护性。

模板是一个包含响应文本的文件，其中包含用占位变量表示的动态部分，其具体值只在请求的上下文中才能知道。使用真实值替换变量，再返回最终得到的响应字符串，这一过程称为*渲染*。为了渲染模板，Flask 使用了一个名为 Jinja2 的强大模板引擎。

3.1 Jinja2模板引擎

形式最简单的 Jinja2 模板就是一个包含响应文本的文件。示例 3-1 是一个 Jinja2 模板，它和示例 2-1 中 `index()` 视图函数的响应一样。

示例 3-1　templates/index.html：Jinja2 模板

```
<h1>Hello World!</h1>
```

示例 2-2 中，视图函数 `user()` 返回的响应中包含一个使用变量表示的动态部分。示例 3-2 实现了这个响应。

示例 3-2　templates/user.html：Jinja2 模板

```
<h1>Hello, {{ name }}!</h1>
```

3.1.1　渲染模板

默认情况下，Flask 在程序文件夹中的 templates 子文件夹中寻找模板。在下一个 hello.py 版本中，要把前面定义的模板保存在 templates 文件夹中，并分别命名为 index.html 和 user.html。

程序中的视图函数需要修改一下，以便渲染这些模板。修改方法参见示例 3-3。

示例 3-3　hello.py：渲染模板

```
from flask import Flask, render_template

# ...

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

Flask 提供的 `render_template` 函数把 Jinja2 模板引擎集成到了程序中。`render_template` 函数的第一个参数是模板的文件名。随后的参数都是键值对，表示模板中变量对应的真实值。在这段代码中，第二个模板收到一个名为 `name` 的变量。

前例中的 `name=name` 是关键字参数，这类关键字参数很常见，但如果你不熟悉它们的话，可能会觉得迷惑且难以理解。左边的“name”表示参数名，就是模板中使用的占位符；右边的“name”是当前作用域中的变量，表示同名参数的值。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 3a` 签出程序的这个版本。

3.1.2 变量

示例 3-2 在模板中使用的 {{ name }} 结构表示一个变量，它是一种特殊的占位符，告诉模板引擎这个位置的值从渲染模板时使用的数据中获取。

Jinja2 能识别所有类型的变量，甚至是一些复杂的类型，例如列表、字典和对象。在模板中使用变量的一些示例如下：

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>
<p>A value from a list: {{ mylist[3] }}.</p>
<p>A value from a list, with a variable index: {{ mylist[myintvar] }}.</p>
<p>A value from an object's method: {{ myobj.somemethod() }}.</p>
```

可以使用过滤器修改变量，过滤器名添加在变量名之后，中间使用竖线分隔。例如，下述模板以首字母大写形式显示变量 name 的值：

```
Hello, {{ name|capitalize }}
```

表 3-1 列出了 Jinja2 提供的部分常用过滤器。

表3-1　Jinja2变量过滤器

过滤器名	说　明
safe	渲染值时不转义
capitalize	把值的首字母转换成大写，其他字母转换成小写
lower	把值转换成小写形式
upper	把值转换成大写形式
title	把值中每个单词的首字母都转换成大写
trim	把值的首尾空格去掉
striptags	渲染之前把值中所有的 HTML 标签都删掉

safe 过滤器值得特别说明一下。默认情况下，出于安全考虑，Jinja2 会转义所有变量。例如，如果一个变量的值为 '<h1>Hello</h1>'，Jinja2 会将其渲染成 '<h1>Hello</h1>'，浏览器能显示这个 h1 元素，但不会进行解释。很多情况下需要显示变量中存储的 HTML 代码，这时就可使用 safe 过滤器。



千万别在不可信的值上使用 safe 过滤器，例如用户在表单中输入的文本。

完整的过滤器列表可在 Jinja2 文档（http://jinja.pocoo.org/docs/templates/#builtin-filters）中查看。

3.1.3 控制结构

Jinja2 提供了多种控制结构，可用来改变模板的渲染流程。本节使用简单的例子介绍其中最有用的控制结构。

下面这个例子展示了如何在模板中使用条件控制语句：

```
{% if user %}
    Hello, {{ user }}!
{% else %}
    Hello, Stranger!
{% endif %}
```

另一种常见需求是在模板中渲染一组元素。下例展示了如何使用 `for` 循环实现这一需求：

```
<ul>
    {% for comment in comments %}
        <li>{{ comment }}</li>
    {% endfor %}
</ul>
```

Jinja2 还支持宏。宏类似于 Python 代码中的函数。例如：

```
{% macro render_comment(comment) %}
    <li>{{ comment }}</li>
{% endmacro %}

<ul>
    {% for comment in comments %}
        {{ render_comment(comment) }}
    {% endfor %}
</ul>
```

为了重复使用宏，我们可以将其保存在单独的文件中，然后在需要使用的模板中导入：

```
{% import 'macros.html' as macros %}
<ul>
    {% for comment in comments %}
        {{ macros.render_comment(comment) }}
    {% endfor %}
</ul>
`
```

需要在多处重复使用的模板代码片段可以写入单独的文件，再包含在所有模板中，以避免重复：

```
{% include 'common.html' %}
```

另一种重复使用代码的强大方式是模板继承，它类似于 Python 代码中的类继承。首先，创建一个名为 base.html 的基模板：

```
<html>
<head>
    {% block head %}
    <title>{% block title %}{% endblock %} - My Application</title>
    {% endblock %}
</head>
<body>
    {% block body %}
    {% endblock %}
</body>
</html>
```

block 标签定义的元素可在衍生模板中修改。在本例中，我们定义了名为 head、title 和 body 的块。注意，title 包含在 head 中。下面这个示例是基模板的衍生模板：

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style>
    </style>
{% endblock %}
{% block body %}
<h1>Hello, World!</h1>
{% endblock %}
```

extends 指令声明这个模板衍生自 base.html。在 extends 指令之后，基模板中的 3 个块被重新定义，模板引擎会将其插入适当的位置。注意新定义的 head 块，在基模板中其内容不是空的，所以使用 super() 获取原来的内容。

稍后会展示这些控制结构的具体用法，让你了解一下它们的工作原理。

3.2　使用Flask-Bootstrap集成Twitter Bootstrap

Bootstrap（http://getbootstrap.com/）是 Twitter 开发的一个开源框架，它提供的用户界面组件可用于创建整洁且具有吸引力的网页，而且这些网页还能兼容所有现代 Web 浏览器。

Bootstrap 是客户端框架，因此不会直接涉及服务器。服务器需要做的只是提供引用了 Bootstrap 层叠样式表（CSS）和 JavaScript 文件的 HTML 响应，并在 HTML、CSS 和 JavaScript 代码中实例化所需组件。这些操作最理想的执行场所就是模板。

要想在程序中集成 Bootstrap，显然要对模板做所有必要的改动。不过，更简单的方法是使用一个名为 Flask-Bootstrap 的 Flask 扩展，简化集成的过程。Flask-Bootstrap 使用 pip 安装：

```
(venv) $ pip install flask-bootstrap
```

Flask 扩展一般都在创建程序实例时初始化。示例 3-4 是 Flask-Bootstrap 的初始化方法。

示例 3-4　hello.py：初始化 Flask-Bootstrap

```
from flask.ext.bootstrap import Bootstrap
# ...
bootstrap = Bootstrap(app)
```

和第 2 章中的 Flask-Script 一样，Flask-Bootstrap 也从 `flask.ext` 命名空间中导入，然后把程序实例传入构造方法进行初始化。

初始化 Flask-Bootstrap 之后，就可以在程序中使用一个包含所有 Bootstrap 文件的基模板。这个模板利用 Jinja2 的模板继承机制，让程序扩展一个具有基本页面结构的基模板，其中就有用来引入 Bootstrap 的元素。示例 3-5 是把 user.html 改写为衍生模板后的新版本。

示例 3-5　templates/user.html：使用 Flask-Bootstrap 的模板

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle"
             data-toggle="collapse" data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="/">Flasky</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="/">Home</a></li>
            </ul>
        </div>
    </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
    <div class="page-header">
        <h1>Hello, {{ name }}!</h1>
    </div>
</div>
{% endblock %}
```

Jinja2 中的 `extends` 指令从 Flask-Bootstrap 中导入 bootstrap/base.html，从而实现模板继承。Flask-Bootstrap 中的基模板提供了一个网页框架，引入了 Bootstrap 中的所有 CSS 和

JavaScript 文件。

基模板中定义了可在衍生模板中重定义的块。`block` 和 `endblock` 指令定义的块中的内容可添加到基模板中。

上面这个 user.html 模板定义了 3 个块，分别名为 `title`、`navbar` 和 `content`。这些块都是基模板提供的，可在衍生模板中重新定义。`title` 块的作用很明显，其中的内容会出现在渲染后的 HTML 文档头部，放在 `<title>` 标签中。`navbar` 和 `content` 这两个块分别表示页面中的导航条和主体内容。

在这个模板中，`navbar` 块使用 Bootstrap 组件定义了一个简单的导航条。`content` 块中有个 `<div>` 容器，其中包含一个页面头部。之前版本的模板中的欢迎信息，现在就放在这个页面头部。改动之后的程序如图 3-1 所示。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 3b` 签出程序的这个版本。Bootstrap 官方文档（http://getbootstrap.com/）是很好的学习资源，有很多可以直接复制粘贴的示例。

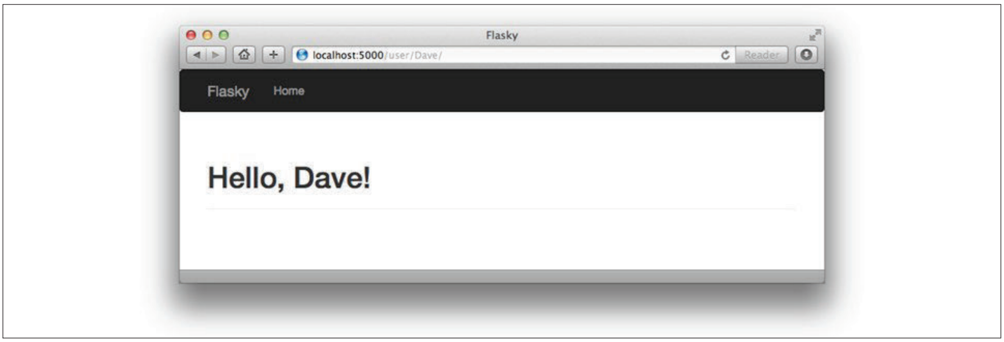


图 3-1　Twitter Bootstrap 模板

Flask-Bootstrap 的 base.html 模板还定义了很多其他块，都可在衍生模板中使用。表 3-2 列出了所有可用的快。

表3-2　Flask-Bootstrap基模板中定义的块

块　　名	说　　明
doc	整个 HTML 文档
html_attribs	`<html>` 标签的属性
html	`<html>` 标签中的内容
head	`<head>` 标签中的内容
title	`<title>` 标签中的内容

（续）

块　　名	说　　明
metas	一组 <meta> 标签
styles	层叠样式表定义
body_attribs	<body> 标签的属性
body	<body> 标签中的内容
navbar	用户定义的导航条
content	用户定义的页面内容
scripts	文档底部的 JavaScript 声明

表 3-2 中的很多块都是 Flask-Bootstrap 自用的，如果直接重定义可能会导致一些问题。例如，Bootstrap 所需的文件在 `styles` 和 `scripts` 块中声明。如果程序需要向已经有内容的块中添加新内容，必须使用 Jinja2 提供的 `super()` 函数。例如，如果要在衍生模板中添加新的 JavaScript 文件，需要这么定义 `scripts` 块：

```
{% block scripts %}
{{ super() }}
<script type="text/javascript" src="my-script.js"></script>
{% endblock %}
```

3.3　自定义错误页面

如果你在浏览器的地址栏中输入了不可用的路由，那么会显示一个状态码为 404 的错误页面。现在这个错误页面太简陋、平庸，而且样式和使用了 Bootstrap 的页面不一致。

像常规路由一样，Flask 允许程序使用基于模板的自定义错误页面。最常见的错误代码有两个：404，客户端请求未知页面或路由时显示；500，有未处理的异常时显示。为这两个错误代码指定自定义处理程序的方式如示例 3-6 所示。

示例 3-6　hello.py：自定义错误页面

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

和视图函数一样，错误处理程序也会返回响应。它们还返回与该错误对应的数字状态码。

错误处理程序中引用的模板也需要编写。这些模板应该和常规页面使用相同的布局，因此要有一个导航条和显示错误消息的页面头部。

编写这些模板最直观的方法是复制 templates/user.html，分别创建 templates/404.html 和

templates/500.html，然后把这两个文件中的页面头部元素改为相应的错误消息。但这种方法会带来很多重复劳动。

Jinja2 的模板继承机制可以帮助我们解决这一问题。Flask-Bootstrap 提供了一个具有页面基本布局的基模板，同样，程序可以定义一个具有更完整页面布局的基模板，其中包含导航条，而页面内容则可留到衍生模板中定义。示例 3-7 展示了 templates/base.html 的内容，这是一个继承自 bootstrap/base.html 的新模板，其中定义了导航条。这个模板本身也可作为其他模板的基模板，例如 templates/user.html、templates/404.html 和 templates/500.html。

示例 3-7　templates/base.html：包含导航条的程序基模板

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle"
             data-toggle="collapse" data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="/">Flasky</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="/">Home</a></li>
            </ul>
        </div>
    </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
    {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

这个模板的 `content` 块中只有一个 `<div>` 容器，其中包含了一个名为 `page_content` 的新的空块，块中的内容由衍生模板定义。

现在，程序使用的模板继承自这个模板，而不直接继承自 Flask-Bootstrap 的基模板。通过继承 templates/base.html 模板编写自定义的 404 错误页面很简单，如示例 3-8 所示。

示例 3-8　templates/404.html：使用模板继承机制自定义 404 错误页面

```
{% extends "base.html" %}

{% block title %}Flasky - Page Not Found{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Not Found</h1>
</div>
{% endblock %}
```

错误页面在浏览器中的显示效果如图 3-2 所示。

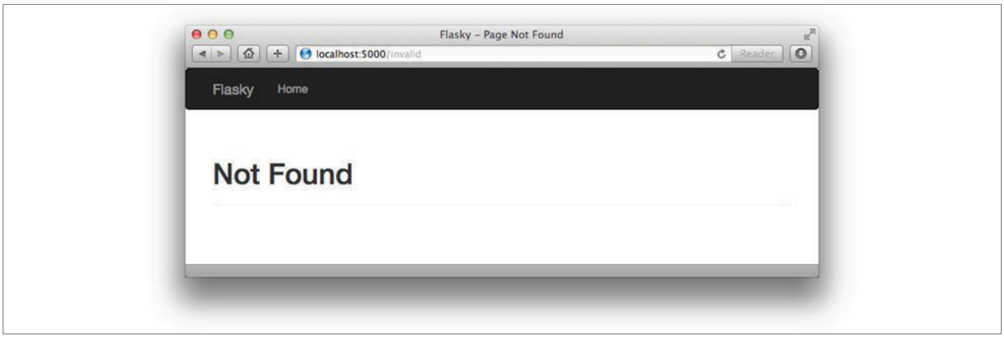


图 3-2　自定义的 404 错误页面

templates/user.html 现在可以通过继承这个基模板来简化内容，如示例 3-9 所示。

示例 3-9　templates/user.html：使用模板继承机制简化页面模板

```
{% extends "base.html" %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {{ name }}!</h1>
</div>
{% endblock %}
```

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 3c` 签出程序的这个版本。

3.4　链接

任何具有多个路由的程序都需要可以连接不同页面的链接，例如导航条。

在模板中直接编写简单路由的 URL 链接不难，但对于包含可变部分的动态路由，在模板中构建正确的 URL 就很困难。而且，直接编写 URL 会对代码中定义的路由产生不必要的依赖关系。如果重新定义路由，模板中的链接可能会失效。

为了避免这些问题，Flask 提供了 `url_for()` 辅助函数，它可以使用程序 URL 映射中保存的信息生成 URL。

`url_for()` 函数最简单的用法是以视图函数名（或者 `app.add_url_route()` 定义路由时使用的端点名）作为参数，返回对应的 URL。例如，在当前版本的 hello.py 程序中调用 `url_for('index')` 得到的结果是 `/`。调用 `url_for('index', _external=True)` 返回的则是绝对地址，在这个示例中是 http://localhost:5000/。

> 生成连接程序内不同路由的链接时，使用相对地址就足够了。如果要生成在浏览器之外使用的链接，则必须使用绝对地址，例如在电子邮件中发送的链接。

使用 `url_for()` 生成动态地址时，将动态部分作为关键字参数传入。例如，`url_for('user', name='john', _external=True)` 的返回结果是 http://localhost:5000/user/john。

传入 `url_for()` 的关键字参数不仅限于动态路由中的参数。函数能将任何额外参数添加到查询字符串中。例如，`url_for('index', page=2)` 的返回结果是 /?page=2。

3.5　静态文件

Web 程序不是仅由 Python 代码和模板组成。大多数程序还会使用静态文件，例如 HTML 代码中引用的图片、JavaScript 源码文件和 CSS。

你可能还记得在第 2 章中检查 hello.py 程序的 URL 映射时，其中有一个 `static` 路由。这是因为对静态文件的引用被当成一个特殊的路由，即 */static/<filename>*。例如，调用 `url_for('static', filename='css/styles.css', _external=True)` 得到的结果是 http://localhost:5000/static/css/styles.css。

默认设置下，Flask 在程序根目录中名为 static 的子目录中寻找静态文件。如果需要，可在 static 文件夹中使用子文件夹存放文件。服务器收到前面那个 URL 后，会生成一个响应，包含文件系统中 static/css/styles.css 文件的内容。

示例 3-10 展示了如何在程序的基模板中放置 favicon.ico 图标。这个图标会显示在浏览器的地址栏中。

示例 3-10　templates/base.html：定义收藏夹图标

```
{% block head %}
{{ super() }}
<link rel="shortcut icon" href="{{ url_for('static', filename = 'favicon.ico') }}"
    type="image/x-icon">
<link rel="icon" href="{{ url_for('static', filename = 'favicon.ico') }}"
    type="image/x-icon">
{% endblock %}
```

图标的声明会插入 `head` 块的末尾。注意如何使用 `super()` 保留基模板中定义的块的原始内容。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 3d` 签出程序的这个版本。

3.6　使用Flask-Moment本地化日期和时间

如果 Web 程序的用户来自世界各地，那么处理日期和时间可不是一个简单的任务。

服务器需要统一时间单位，这和用户所在的地理位置无关，所以一般使用协调世界时（Coordinated Universal Time，UTC）。不过用户看到 UTC 格式的时间会感到困惑，他们更希望看到当地时间，而且采用当地惯用的格式。

要想在服务器上只使用 UTC 时间，一个优雅的解决方案是，把时间单位发送给 Web 浏览器，转换成当地时间，然后渲染。Web 浏览器可以更好地完成这一任务，因为它能获取用户电脑中的时区和区域设置。

有一个使用 JavaScript 开发的优秀客户端开源代码库，名为 moment.js（http://momentjs.com/），它可以在浏览器中渲染日期和时间。Flask-Moment 是一个 Flask 程序扩展，能把 moment.js 集成到 Jinja2 模板中。Flask-Moment 可以使用 `pip` 安装：

```
(venv) $ pip install flask-moment
```

这个扩展的初始化方法如示例 3-11 所示。

示例 3-11　hello.py：初始化 Flask-Moment

```
from flask.ext.moment import Moment
moment = Moment(app)
```

除了 moment.js，Flask-Moment 还依赖 jquery.js。要在 HTML 文档的某个地方引入这两个库，可以直接引入，这样可以选择使用哪个版本，也可使用扩展提供的辅助函数，从内容分发网络（Content Delivery Network，CDN）中引入通过测试的版本。Bootstrap 已经引入了 jquery.js，因此只需引入 moment.js 即可。示例 3-12 展示了如何在基模板的 `scripts` 块中引入这个库。

示例 3-12　templates/base.html：引入 moment.js 库

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{% endblock %}
```

为了处理时间戳，Flask-Moment 向模板开放了 `moment` 类。示例 3-13 中的代码把变量 `current_time` 传入模板进行渲染。

示例 3-13　hello.py：加入一个 datetime 变量

```
from datetime import datetime

@app.route('/')
def index():
    return render_template('index.html',
                           current_time=datetime.utcnow())
```

示例 3-14 展示了如何在模板中渲染 `current_time`。

代码 3-14　templates/index.html：使用 Flask-Moment 渲染时间戳

```
<p>The local date and time is {{ moment(current_time).format('LLL') }}.</p>
<p>That was {{ moment(current_time).fromNow(refresh=True) }}</p>
```

`format('LLL')` 根据客户端电脑中的时区和区域设置渲染日期和时间。参数决定了渲染的方式，`'L'` 到 `'LLLL'` 分别对应不同的复杂度。`format()` 函数还可接受自定义的格式说明符。

第二行中的 `fromNow()` 渲染相对时间戳，而且会随着时间的推移自动刷新显示的时间。这个时间戳最开始显示为“a few seconds ago”，但指定 `refresh` 参数后，其内容会随着时间的推移而更新。如果一直待在这个页面，几分钟后，会看到显示的文本变成“a minute ago”“2 minutes ago”等。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 3e` 签出程序的这个版本。

Flask-Moment 实现了 moment.js 中的 `format()`、`fromNow()`、`fromTime()`、`calendar()`、`valueOf()` 和 `unix()` 方法。你可查阅文档（http://momentjs.com/docs/#/displaying/）学习 moment.js 提供的全

部格式化选项。

> Flask-Monet 假定服务器端程序处理的时间戳是“纯正的”`datetime` 对象，且使用 UTC 表示。关于纯正和细致的日期和时间对象[1]的说明，请阅读标准库中 `datetime` 包的文档（https://docs.python.org/2/library/datetime.html）。

Flask-Moment 渲染的时间戳可实现多种语言的本地化。语言可在模板中选择，把语言代码传给 `lang()` 函数即可：

```
{{ moment.lang('es') }}
```

使用本章介绍的技术，你应该能为程序编写出现代化且用户友好的网页。下一章将介绍本章没有涉及的一个模板功能，即如何通过 Web 表单和用户交互。

译注 1：纯正的时间戳，英文为 navie time，指不包含时区的时间戳；细致的时间戳，英文为 aware time，指包含时区的时间戳。

第4章 Web表单

第 2 章中介绍的请求对象包含客户端发出的所有请求信息。其中，`request.form` 能获取 `POST` 请求中提交的表单数据。

尽管 Flask 的请求对象提供的信息足够用于处理 Web 表单，但有些任务很单调，而且要重复操作。比如，生成表单的 HTML 代码和验证提交的表单数据。

Flask-WTF（http://pythonhosted.org/Flask-WTF/）扩展可以把处理 Web 表单的过程变成一种愉悦的体验。这个扩展对独立的 WTForms（http://wtforms.simplecodes.com）包进行了包装，方便集成到 Flask 程序中。

Flask-WTF 及其依赖可使用 `pip` 安装：

```
(venv) $ pip install flask-wtf
```

4.1 跨站请求伪造保护

默认情况下，Flask-WTF 能保护所有表单免受跨站请求伪造（Cross-Site Request Forgery，CSRF）的攻击。恶意网站把请求发送到被攻击者已登录的其他网站时就会引发 CSRF 攻击。

为了实现 CSRF 保护，Flask-WTF 需要程序设置一个密钥。Flask-WTF 使用这个密钥生成加密令牌，再用令牌验证请求中表单数据的真伪。设置密钥的方法如示例 4-1 所示。

示例 4-1 hello.py：设置 Flask-WTF

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess string'
```

app.config 字典可用来存储框架、扩展和程序本身的配置变量。使用标准的字典句法就能把配置值添加到 app.config 对象中。这个对象还提供了一些方法，可以从文件或环境中导入配置值。

SECRET_KEY 配置变量是通用密钥，可在 Flask 和多个第三方扩展中使用。如其名所示，加密的强度取决于变量值的机密程度。不同的程序要使用不同的密钥，而且要保证其他人不知道你所用的字符串。

> 为了增强安全性，密钥不应该直接写入代码，而要保存在环境变量中。这一技术会在第 7 章介绍。

4.2 表单类

使用 Flask-WTF 时，每个 Web 表单都由一个继承自 Form 的类表示。这个类定义表单中的一组字段，每个字段都用对象表示。字段对象可附属一个或多个验证函数。验证函数用来验证用户提交的输入值是否符合要求。

示例 4-2 是一个简单的 Web 表单，包含一个文本字段和一个提交按钮。

示例 4-2 hello.py：定义表单类

```
from flask.ext.wtf import Form
from wtforms import StringField, SubmitField
from wtforms.validators import Required

class NameForm(Form):
    name = StringField('What is your name?', validators=[Required()])
    submit = SubmitField('Submit')
```

这个表单中的字段都定义为类变量，类变量的值是相应字段类型的对象。在这个示例中，NameForm 表单中有一个名为 name 的文本字段和一个名为 submit 的提交按钮。StringField 类表示属性为 type="text" 的 <input> 元素。SubmitField 类表示属性为 type="submit" 的 <input> 元素。字段构造函数的第一个参数是把表单渲染成 HTML 时使用的标号。

StringField 构造函数中的可选参数 validators 指定一个由验证函数组成的列表，在接受用户提交的数据之前验证数据。验证函数 Required() 确保提交的字段不为空。

> Form 基类由 Flask-WTF 扩展定义，所以从 flask.ext.wtf 中导入。字段和验证函数却可以直接从 WTForms 包中导入。

WTForms 支持的 HTML 标准字段如表 4-1 所示。

表4-1　WTForms支持的HTML标准字段

字段类型	说　　明
StringField	文本字段
TextAreaField	多行文本字段
PasswordField	密码文本字段
HiddenField	隐藏文本字段
DateField	文本字段，值为 datetime.date 格式
DateTimeField	文本字段，值为 datetime.datetime 格式
IntegerField	文本字段，值为整数
DecimalField	文本字段，值为 decimal.Decimal
FloatField	文本字段，值为浮点数
BooleanField	复选框，值为 True 和 False
RadioField	一组单选框
SelectField	下拉列表
SelectMultipleField	下拉列表，可选择多个值
FileField	文件上传字段
SubmitField	表单提交按钮
FormField	把表单作为字段嵌入另一个表单
FieldList	一组指定类型的字段

WTForms 内建的验证函数如表 4-2 所示。

表4-2　WTForms验证函数

验证函数	说　　明
Email	验证电子邮件地址
EqualTo	比较两个字段的值；常用于要求输入两次密码进行确认的情况
IPAddress	验证 IPv4 网络地址
Length	验证输入字符串的长度
NumberRange	验证输入的值在数字范围内
Optional	无输入值时跳过其他验证函数
Required	确保字段中有数据
Regexp	使用正则表达式验证输入值
URL	验证 URL
AnyOf	确保输入值在可选值列表中
NoneOf	确保输入值不在可选值列表中

4.3　把表单渲染成HTML

表单字段是可调用的，在模板中调用后会渲染成 HTML。假设视图函数把一个 NameForm 实

例通过参数 form 传入模板，在模板中可以生成一个简单的表单，如下所示：

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }}
</form>
```

当然，这个表单还很简陋。要想改进表单的外观，可以把参数传入渲染字段的函数，传入的参数会被转换成字段的 HTML 属性。例如，可以为字段指定 id 或 class 属性，然后定义 CSS 样式：

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(id='my-text-field') }}
    {{ form.submit() }}
</form>
```

即便能指定 HTML 属性，但按照这种方式渲染表单的工作量还是很大，所以在条件允许的情况下最好能使用 Bootstrap 中的表单样式。Flask-Bootstrap 提供了一个非常高端的辅助函数，可以使用 Bootstrap 中预先定义好的表单样式渲染整个 Flask-WTF 表单，而这些操作只需一次调用即可完成。使用 Flask-Bootstrap，上述表单可使用下面的方式渲染：

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

import 指令的使用方法和普通 Python 代码一样，允许导入模板中的元素并用在多个模板中。导入的 bootstrap/wtf.html 文件中定义了一个使用 Bootstrap 渲染 Falsk-WTF 表单对象的辅助函数。wtf.quick_form() 函数的参数为 Flask-WTF 表单对象，使用 Bootstrap 的默认样式渲染传入的表单。hello.py 的完整模板如示例 4-3 所示。

示例 4-3 templates/index.html：使用 Flask-WTF 和 Flask-Bootstrap 渲染表单

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

模板的内容区现在有两部分。第一部分是页面头部，显示欢迎消息。这里用到了一个模板条件语句。Jinja2 中的条件语句格式为 {% if condition %}...{% else %}...{% endif %}。如果条件的计算结果为 True，那么渲染 if 和 else 指令之间的值。如果条件的计算结果为 False，则渲染 else 和 endif 指令之间的值。在这个例子中，如果没有定义模板变量 name，

则会渲染字符串“Hello, Stranger!”。内容区的第二部分使用 wtf.quick_form() 函数渲染 NameForm 对象。

4.4 在视图函数中处理表单

在新版 hello.py 中，视图函数 index() 不仅要渲染表单，还要接收表单中的数据。示例 4-4 是更新后的 index() 视图函数。

示例 4-4 hello.py：路由方法

```
@app.route('/', methods=['GET', 'POST'])
def index():
    name = None
    form = NameForm()
    if form.validate_on_submit():
        name = form.name.data
        form.name.data = ''
    return render_template('index.html', form=form, name=name)
```

app.route 修饰器中添加的 methods 参数告诉 Flask 在 URL 映射中把这个视图函数注册为 GET 和 POST 请求的处理程序。如果没指定 methods 参数，就只把视图函数注册为 GET 请求的处理程序。

把 POST 加入方法列表很有必要，因为将提交表单作为 POST 请求进行处理更加便利。表单也可作为 GET 请求提交，不过 GET 请求没有主体，提交的数据以查询字符串的形式附加到 URL 中，可在浏览器的地址栏中看到。基于这个以及其他多个原因，提交表单大都作为 POST 请求进行处理。

局部变量 name 用来存放表单中输入的有效名字，如果没有输入，其值为 None。如上述代码所示，在视图函数中创建一个 NameForm 类实例用于表示表单。提交表单后，如果数据能被所有验证函数接受，那么 validate_on_submit() 方法的返回值为 True，否则返回 False。这个函数的返回值决定是重新渲染表单还是处理表单提交的数据。

用户第一次访问程序时，服务器会收到一个没有表单数据的 GET 请求，所以 validate_on_submit() 将返回 False。if 语句的内容将被跳过，通过渲染模板处理请求，并传入表单对象和值为 None 的 name 变量作为参数。用户会看到浏览器中显示了一个表单。

用户提交表单后，服务器收到一个包含数据的 POST 请求。validate_on_submit() 会调用 name 字段上附属的 Required() 验证函数。如果名字不为空，就能通过验证，validate_on_submit() 返回 True。现在，用户输入的名字可通过字段的 data 属性获取。在 if 语句中，把名字赋值给局部变量 name，然后再把 data 属性设为空字符串，从而清空表单字段。最后一行调用 render_template() 函数渲染模板，但这一次参数 name 的值为表单中输入的名字，因此会显示一个针对该用户的欢迎消息。

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 4a` 签出程序的这个版本。

图 4-1 是用户首次访问网站时浏览器显示的表单。用户提交名字后，程序会生成一个针对该用户的欢迎消息。欢迎消息下方还是会显示这个表单，以便用户输入新名字。图 4-2 显示了此时程序的样子。

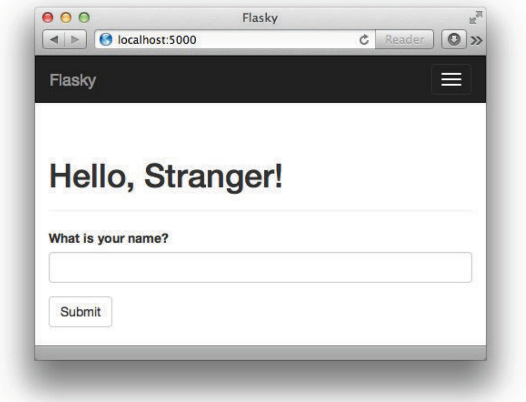


图 4-1　Flask-WTF Web 表单

如果用户提交表单之前没有输入名字，`Required()` 验证函数会捕获这个错误，如图 4-3 所示。注意一下扩展自动提供了多少功能。这说明像 Flask-WTF 和 Flask-Bootstrap 这样设计良好的扩展能让程序具有强大的功能。

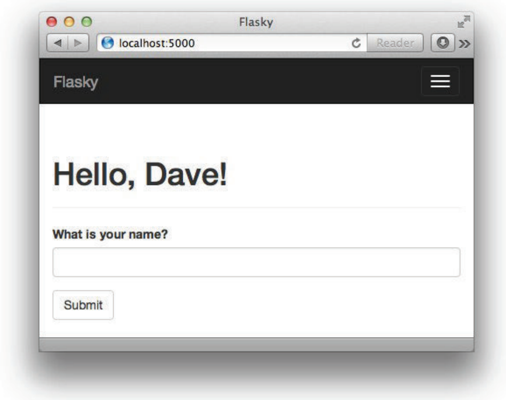


图 4-2　提交后显示的 Web 表单

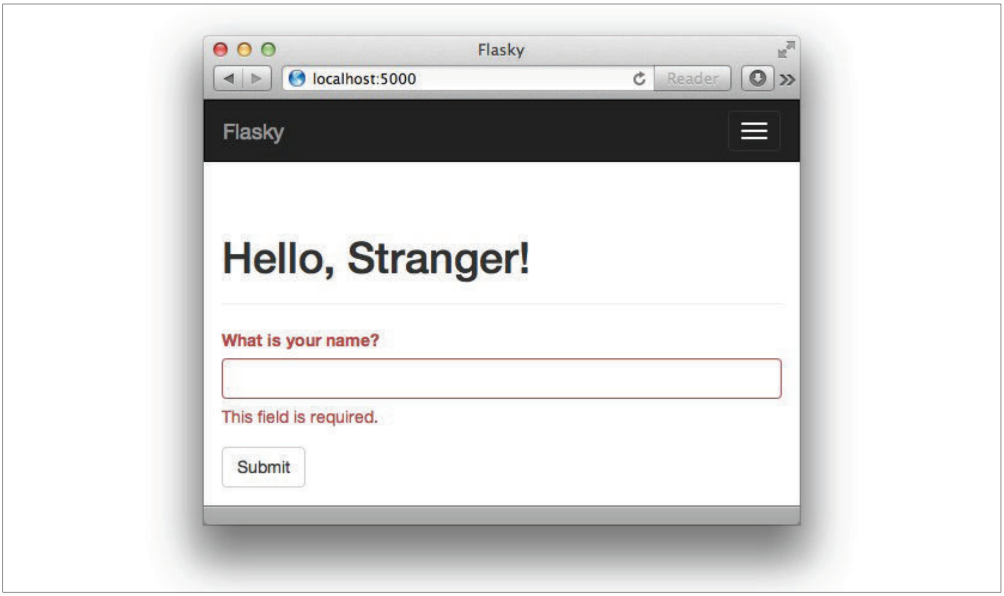


图 4-3　验证失败后显示的 Web 表单

4.5　重定向和用户会话

最新版的 hello.py 存在一个可用性问题。用户输入名字后提交表单，然后点击浏览器的刷新按钮，会看到一个莫名其妙的警告，要求在再次提交表单之前进行确认。之所以出现这种情况，是因为刷新页面时浏览器会重新发送之前已经发送过的最后一个请求。如果这个请求是一个包含表单数据的 `POST` 请求，刷新页面后会再次提交表单。大多数情况下，这并不是理想的处理方式。

很多用户都不理解浏览器发出的这个警告。基于这个原因，最好别让 Web 程序把 `POST` 请求作为浏览器发送的最后一个请求。

这种需求的实现方式是，使用*重定向*作为 `POST` 请求的响应，而不是使用常规响应。重定向是一种特殊的响应，响应内容是 URL，而不是包含 HTML 代码的字符串。浏览器收到这种响应时，会向重定向的 URL 发起 `GET` 请求，显示页面的内容。这个页面的加载可能要多花几微秒，因为要先把第二个请求发给服务器。除此之外，用户不会察觉到有什么不同。现在，最后一个请求是 `GET` 请求，所以刷新命令能像预期的那样正常使用了。这个技巧称为 *Post/ 重定向 /Get 模式*。

但这种方法会带来另一个问题。程序处理 `POST` 请求时，使用 `form.name.data` 获取用户输入的名字，可是一旦这个请求结束，数据也就丢失了。因为这个 `POST` 请求使用重定向处理，所以程序需要保存输入的名字，这样重定向后的请求才能获得并使用这个名字，从而

构建真正的响应。

程序可以把数据存储在*用户会话*中，在请求之间“记住”数据。用户会话是一种私有存储，存在于每个连接到服务器的客户端中。我们在第 2 章介绍过用户会话，它是请求上下文中的变量，名为 `session`，像标准的 Python 字典一样操作。

> 默认情况下，用户会话保存在客户端 cookie 中，使用设置的 `SECRET_KEY` 进行加密签名。如果篡改了 cookie 中的内容，签名就会失效，会话也会随之失效。

示例 4-5 是 `index()` 视图函数的新版本，实现了重定向和用户会话。

示例 4-5 hello.py：重定向和用户会话

```
from flask import Flask, render_template, session, redirect, url_for

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'))
```

在程序的前一个版本中，局部变量 `name` 被用于存储用户在表单中输入的名字。这个变量现在保存在用户会话中，即 `session['name']`，所以在两次请求之间也能记住输入的值。

现在，包含合法表单数据的请求最后会调用 `redirect()` 函数。`redirect()` 是个辅助函数，用来生成 HTTP 重定向响应。`redirect()` 函数的参数是重定向的 URL，这里使用的重定向 URL 是程序的根地址，因此重定向响应本可以写得更简单一些，写成 `redirect('/')`，但却会使用 Flask 提供的 URL 生成函数 `url_for()`。推荐使用 `url_for()` 生成 URL，因为这个函数使用 URL 映射生成 URL，从而保证 URL 和定义的路由兼容，而且修改路由名字后依然可用。

`url_for()` 函数的第一个且唯一必须指定的参数是*端点名*，即路由的内部名字。默认情况下，路由的端点是相应视图函数的名字。在这个示例中，处理根地址的视图函数是 `index()`，因此传给 `url_for()` 函数的名字是 `index`。

最后一处改动位于 `render_function()` 函数中，使用 `session.get('name')` 直接从会话中读取 `name` 参数的值。和普通的字典一样，这里使用 `get()` 获取字典中键对应的值以避免未找到键的异常情况，因为对于不存在的键，`get()` 会返回默认值 `None`。

如果你从 GitHub 上克隆了这个程序的 Git 仓库，可以执行 `git checkout 4b` 签出程序的这个版本。

使用这个版本的程序时，刷新浏览器页面，你看到的新页面就和预期一样了。

4.6 Flash消息

请求完成后，有时需要让用户知道状态发生了变化。这里可以使用确认消息、警告或者错误提醒。一个典型例子是，用户提交了有一项错误的登录表单后，服务器发回的响应重新渲染了登录表单，并在表单上面显示一个消息，提示用户用户名或密码错误。

这种功能是 Flask 的核心特性。如示例 4-6 所示，`flash()` 函数可实现这种效果。

示例 4-6　hello.py：Flash 消息

```
from flask import Flask, render_template, session, redirect, url_for, flash

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        old_name = session.get('name')
        if old_name is not None and old_name != form.name.data:
            flash('Looks like you have changed your name!')
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html',
        form = form, name = session.get('name'))
```

在这个示例中，每次提交的名字都会和存储在用户会话中的名字进行比较，而会话中存储的名字是前一次在这个表单中提交的数据。如果两个名字不一样，就会调用 `flash()` 函数，在发给客户端的下一个响应中显示一个消息。

仅调用 `flash()` 函数并不能把消息显示出来，程序使用的模板要渲染这些消息。最好在基模板中渲染 Flash 消息，因为这样所有页面都能使用这些消息。Flask 把 `get_flashed_messages()` 函数开放给模板，用来获取并渲染消息，如示例 4-7 所示。

示例 4-7　templates/base.html：渲染 Flash 消息

```
{% block content %}
<div class="container">
    {% for message in get_flashed_messages() %}
    <div class="alert alert-warning">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        {{ message }}
```

```
        </div>
        {% endfor %}

        {% block page_content %}{% endblock %}
    </div>
    {% endblock %}
```

在这个示例中，使用 Bootstrap 提供的警报 CSS 样式渲染警告消息（如图 4-4 所示）。

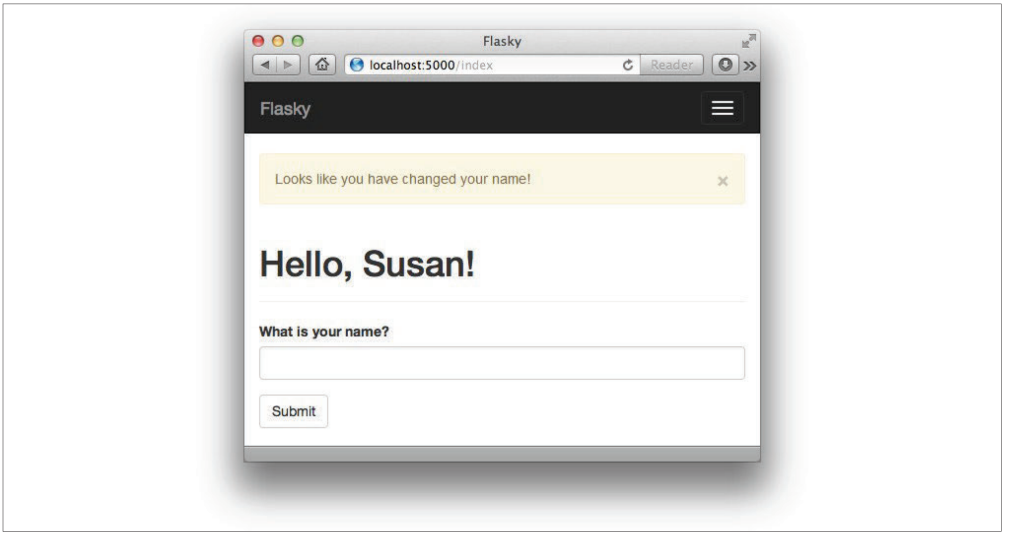
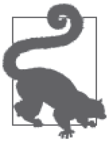


图 4-4　Flash 消息

在模板中使用循环是因为在之前的请求循环中每次调用 `flash()` 函数时都会生成一个消息，所以可能有多个消息在排队等待显示。`get_flashed_messages()` 函数获取的消息在下次调用时不会再次返回，因此 Flash 消息只显示一次，然后就消失了。



如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 4c` 签出程序的这个版本。

从 Web 表单中获取用户输入的数据是大多数程序都需要的功能，把数据保存在永久存储器中也是一样。下一章将介绍如何在 Flask 中使用数据库。

第 5 章

数据库

数据库按照一定规则保存程序数据，程序再发起*查询*取回所需的数据。Web 程序最常用基于*关系*模型的数据库，这种数据库也称为 SQL 数据库，因为它们使用结构化查询语言。不过最近几年*文档数据库*和*键值对数据库*成了流行的替代选择，这两种数据库合称 NoSQL 数据库。

5.1　SQL数据库

关系型数据库把数据存储在*表*中，表模拟程序中不同的实体。例如，订单管理程序的数据库中可能有表 `customers`、`products` 和 `orders`。

表的*列*数是固定的，*行*数是可变的。列定义表所表示的实体的数据属性。例如，`customers` 表中可能有 `name`、`address`、`phone` 等列。表中的行定义各列对应的真实数据。

表中有个特殊的列，称为*主键*，其值为表中各行的唯一标识符。表中还可以有称为*外键*的列，引用同一个表或不同表中某行的主键。行之间的这种联系称为*关系*，这是关系型数据库模型的基础。

图 5-1 展示了一个简单数据库的关系图。这个数据库中有两个表，分别存储用户和用户角色。连接两个表的线代表两个表之间的关系。

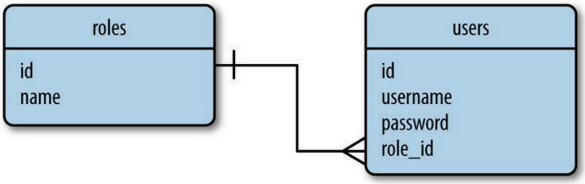


图 5-1　关系型数据库示例

在这个数据库关系图中，`roles` 表存储所有可用的用户角色，每个角色都使用一个唯一的 `id` 值（即表的主键）进行标识。`users` 表包含用户列表，每个用户也有唯一的 `id` 值。除了 `id` 主键之外，`roles` 表中还有 `name` 列，`users` 表中还有 `username` 列和 `password` 列。`users` 表中的 `role_id` 列是外键，引用角色的 `id`，通过这种方式为每个用户指定角色。

从这个例子可以看出，关系型数据库存储数据很高效，而且避免了重复。将这个数据库中的用户角色重命名也很简单，因为角色名只出现在一个地方。一旦在 `roles` 表中修改完角色名，所有通过 `role_id` 引用这个角色的用户都能立即看到更新。

但从另一方面来看，把数据分别存放在多个表中还是很复杂的。生成一个包含角色的用户列表会遇到一个小问题，因为在此之前要分别从两个表中读取用户和用户角色，再将其联结起来。关系型数据库引擎为联结操作提供了必要的支持。

5.2　NoSQL数据库

所有不遵循上节所述的关系模型的数据库统称为 NoSQL *数据库*。NoSQL 数据库一般使用*集合*代替表，使用*文档*代替记录。NoSQL 数据库采用的设计方式使联结变得困难，所以大多数数据库根本不支持这种操作。对于结构如图 5-1 所示的 NoSQL 数据库，若要列出各用户及其角色，就需要在程序中执行联结操作，即先读取每个用户的 `role_id`，再在 `roles` 表中搜索对应的记录。

NoSQL 数据库更适合设计成如图 5-2 所示的结构。这是执行*反规范化*操作得到的结果，它减少了表的数量，却增加了数据重复量。

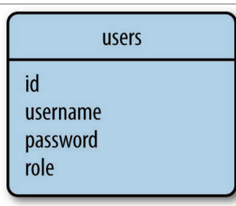


图 5-2　NoSQL 数据库示例

这种结构的数据库要把角色名存储在每个用户中。如此一来，将角色重命名的操作就变得很耗时，可能需要更新大量文档。

使用 NoSQL 数据库当然也有好处。数据重复可以提升查询速度。列出用户及其角色的操作很简单，因为无需联结。

5.3 使用SQL还是NoSQL

SQL 数据库擅于用高效且紧凑的形式存储结构化数据。这种数据库需要花费大量精力保证数据的一致性。NoSQL 数据库放宽了对这种一致性的要求，从而获得性能上的优势。

对不同类型数据库的全面分析、对比超出了本书范畴。对中小型程序来说，SQL 和 NoSQL 数据库都是很好的选择，而且性能相当。

5.4 Python数据库框架

大多数的数据库引擎都有对应的 Python 包，包括开源包和商业包。Flask 并不限制你使用何种类型的数据库包，因此可以根据自己的喜好选择使用 MySQL、Postgres、SQLite、Redis、MongoDB 或者 CouchDB。

如果这些都无法满足需求，还有一些数据库抽象层代码包供选择，例如 SQLAlchemy 和 MongoEngine。你可以使用这些抽象包直接处理高等级的 Python 对象，而不用处理如表、文档或查询语言此类的数据库实体。

选择数据库框架时，你要考虑很多因素。

易用性

如果直接比较数据库引擎和数据库抽象层，显然后者取胜。抽象层，也称为对象关系映射（Object-Relational Mapper，ORM）或对象文档映射（Object-Document Mapper，ODM），在用户不知觉的情况下把高层的面向对象操作转换成低层的数据库指令。

性能

ORM 和 ODM 把对象业务转换成数据库业务会有一定的损耗。大多数情况下，这种性能的降低微不足道，但也不一定都是如此。一般情况下，ORM 和 ODM 对生产率的提升远远超过了这一丁点儿的性能降低，所以性能降低这个理由不足以说服用户完全放弃 ORM 和 ODM。真正的关键点在于如何选择一个能直接操作低层数据库的抽象层，以防特定的操作需要直接使用数据库原生指令优化。

可移植性

选择数据库时，必须考虑其是否能在你的开发平台和生产平台中使用。例如，如果你打算利用云平台托管程序，就要知道这个云服务提供了哪些数据库可供选择。

可移植性还针对 ORM 和 ODM。尽管有些框架只为一种数据库引擎提供抽象层，但其他框架可能做了更高层的抽象，它们支持不同的数据库引擎，而且都使用相同的面向对象接口。SQLAlchemy ORM 就是一个很好的例子，它支持很多关系型数据库引擎，包括流行的 MySQL、Postgres 和 SQLite。

FLask集成度

选择框架时，你不一定非得选择已经集成了 Flask 的框架，但选择这些框架可以节省你编写集成代码的时间。使用集成了 Flask 的框架可以简化配置和操作，所以专门为 Flask 开发的扩展是你的首选。

基于以上因素，本书选择使用的数据库框架是 Flask-SQLAlchemy（http://pythonhosted.org/Flask-SQLAlchemy/），这个 Flask 扩展包装了 SQLAlchemy（http://www.sqlalchemy.org/）框架。

5.5 使用Flask-SQLAlchemy管理数据库

Flask-SQLAlchemy 是一个 Flask 扩展，简化了在 Flask 程序中使用 SQLAlchemy 的操作。SQLAlchemy 是一个很强大的关系型数据库框架，支持多种数据库后台。SQLAlchemy 提供了高层 ORM，也提供了使用数据库原生 SQL 的低层功能。

和其他大多数扩展一样，Flask-SQLAlchemy 也使用 `pip` 安装：

```
(venv) $ pip install flask-sqlalchemy
```

在 Flask-SQLAlchemy 中，数据库使用 URL 指定。最流行的数据库引擎采用的数据库 URL 格式如表 5-1 所示。

表5-1　FLask-SQLAlchemy数据库URL

数据库引擎	URL
MySQL	*mysql://username:password@hostname/database*
Postgres	*postgresql://username:password@hostname/database*
SQLite（Unix）	*sqlite:////absolute/path/to/database*
SQLite（Windows）	*sqlite:///c:/absolute/path/to/database*

在这些 URL 中，*hostname* 表示 MySQL 服务所在的主机，可以是本地主机（*localhost*），也可以是远程服务器。数据库服务器上可以托管多个数据库，因此 *database* 表示要使用的数据库名。如果数据库需要进行认证，*username* 和 *password* 表示数据库用户密令。

SQLite 数据库不需要使用服务器，因此不用指定 *hostname*、*username* 和 *password*。URL 中的 *database* 是硬盘上文件的文件名。

程序使用的数据库 URL 必须保存到 Flask 配置对象的 SQLALCHEMY_DATABASE_URI 键中。配置对象中还有一个很有用的选项，即 SQLALCHEMY_COMMIT_ON_TEARDOWN 键，将其设为 True 时，每次请求结束后都会自动提交数据库中的变动。其他配置选项的作用请参阅 Flask-SQLAlchemy 的文档。示例 5-1 展示了如何初始化及配置一个简单的 SQLite 数据库。

示例 5-1　hello.py：配置数据库

```
from flask.ext.sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.path.dirname(__file__))

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =\
    'sqlite:///' + os.path.join(basedir, 'data.sqlite')
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] = True

db = SQLAlchemy(app)
```

db 对象是 SQLAlchemy 类的实例，表示程序使用的数据库，同时还获得了 Flask-SQLAlchemy 提供的所有功能。

5.6　定义模型

模型这个术语表示程序使用的持久化实体。在 ORM 中，模型一般是一个 Python 类，类中的属性对应数据库表中的列。

Flask-SQLAlchemy 创建的数据库实例为模型提供了一个基类以及一系列辅助类和辅助函数，可用于定义模型的结构。图 5-1 中的 roles 表和 users 表可定义为模型 Role 和 User，如示例 5-2 所示。

示例 5-2　hello.py：定义 Role 和 User 模型

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)

    def __repr__(self):
        return '<Role %r>' % self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)

    def __repr__(self):
        return '<User %r>' % self.username
```

类变量 __tablename__ 定义在数据库中使用的表名。如果没有定义 __tablename__，Flask-

SQLAlchemy 会使用一个默认名字，但默认的表名没有遵守使用复数形式进行命名的约定，所以最好由我们自己来指定表名。其余的类变量都是该模型的属性，被定义为 `db.Column` 类的实例。

`db.Column` 类构造函数的第一个参数是数据库列和模型属性的类型。表 5-2 列出了一些可用的列类型以及在模型中使用的 Python 类型。

表5-2　最常用的SQLAlchemy列类型

类型名	Python类型	说　　明
Integer	`int`	普通整数，一般是 32 位
SmallInteger	`int`	取值范围小的整数，一般是 16 位
BigInteger	`int` 或 `long`	不限制精度的整数
Float	`float`	浮点数
Numeric	`decimal.Decimal`	定点数
String	`str`	变长字符串
Text	`str`	变长字符串，对较长或不限长度的字符串做了优化
Unicode	`unicode`	变长 Unicode 字符串
UnicodeText	`unicode`	变长 Unicode 字符串，对较长或不限长度的字符串做了优化
Boolean	`bool`	布尔值
Date	`datetime.date`	日期
Time	`datetime.time`	时间
DateTime	`datetime.datetime`	日期和时间
Interval	`datetime.timedelta`	时间间隔
Enum	`str`	一组字符串
PickleType	任何 Python 对象	自动使用 Pickle 序列化
LargeBinary	`str`	二进制文件

`db.Column` 中其余的参数指定属性的配置选项。表 5-3 列出了一些可用选项。

表5-3　最常使用的SQLAlchemy列选项

选项名	说　　明
`primary_key`	如果设为 `True`，这列就是表的主键
`unique`	如果设为 `True`，这列不允许出现重复的值
`index`	如果设为 `True`，为这列创建索引，提升查询效率
`nullable`	如果设为 `True`，这列允许使用空值；如果设为 `False`，这列不允许使用空值
`default`	为这列定义默认值

Flask-SQLAlchemy 要求每个模型都要定义主键，这一列经常命名为 `id`。

虽然没有强制要求，但这两个模型都定义了 `__repr()__` 方法，返回一个具有可读性的字符串表示模型，可在调试和测试时使用。

5.7　关系

关系型数据库使用关系把不同表中的行联系起来。图 5-1 所示的关系图表示用户和角色之间的一种简单关系。这是角色到用户的一对多关系，因为一个角色可属于多个用户，而每个用户都只能有一个角色。

图 5-1 中的一对多关系在模型类中的表示方法如示例 5-3 所示。

示例 5-3　hello.py：关系

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role')

class User(db.Model):
    # ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

如图 5-1 所示，关系使用 `users` 表中的外键连接了两行。添加到 `User` 模型中的 `role_id` 列被定义为外键，就是这个外键建立起了关系。传给 `db.ForeignKey()` 的参数 `'roles.id'` 表明，这列的值是 `roles` 表中行的 `id` 值。

添加到 `Role` 模型中的 `users` 属性代表这个关系的面向对象视角。对于一个 `Role` 类的实例，其 `users` 属性将返回与角色相关联的用户组成的列表。`db.relationship()` 的第一个参数表明这个关系的另一端是哪个模型。如果模型类尚未定义，可使用字符串形式指定。

`db.relationship()` 中的 `backref` 参数向 `User` 模型中添加一个 `role` 属性，从而定义反向关系。这一属性可替代 `role_id` 访问 `Role` 模型，此时获取的是模型对象，而不是外键的值。

大多数情况下，`db.relationship()` 都能自行找到关系中的外键，但有时却无法决定把哪一列作为外键。例如，如果 `User` 模型中有两个或以上的列定义为 `Role` 模型的外键，SQLAlchemy 就不知道该使用哪列。如果无法决定外键，你就要为 `db.relationship()` 提供额外参数，从而确定所用外键。表 5-4 列出了定义关系时常用的配置选项。

表5-4　常用的SQLAlchemy关系选项

选项名	说　明
`backref`	在关系的另一个模型中添加反向引用
`primaryjoin`	明确指定两个模型之间使用的联结条件。只在模棱两可的关系中需要指定
`lazy`	指定如何加载相关记录。可选值有 `select`（首次访问时按需加载）、`immediate`（源对象加载后就加载）、`joined`（加载记录，但使用联结）、`subquery`（立即加载，但使用子查询），`noload`（永不加载）和 `dynamic`（不加载记录，但提供加载记录的查询）

（续）

选项名	说　明
uselist	如果设为 Fales，不使用列表，而使用标量值
order_by	指定关系中记录的排序方式
secondary	指定多对多关系中关系表的名字
secondaryjoin	SQLAlchemy 无法自行决定时，指定多对多关系中的二级联结条件

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 5a` 签出程序的这个版本。

除了一对多之外，还有几种其他的关系类型。一对一关系可以用前面介绍的一对多关系表示，但调用 `db.relationship()` 时要把 `uselist` 设为 `False`，把“多”变成“一”。多对一关系也可使用一对多表示，对调两个表即可，或者把外键和 `db.relationship()` 都放在“多”这一侧。最复杂的关系类型是多对多，需要用到第三张表，这个表称为*关系表*。你将在第 12 章学习多对多关系。

5.8 数据库操作

现在模型已经按照图 5-1 所示的数据库关系图完成配置，可以随时使用了。学习如何使用模型的最好方法是在 Python shell 中实际操作。接下来的几节将介绍最常用的数据库操作。

5.8.1 创建表

首先，我们要让 Flask-SQLAlchemy 根据模型类创建数据库。方法是使用 `db.create_all()` 函数：

```
(venv) $ python hello.py shell
>>> from hello import db
>>> db.create_all()
```

如果你查看程序目录，会发现新建了一个名为 data.sqlite 的文件。这个 SQLite 数据库文件的名字就是在配置中指定的。如果数据库表已经存在于数据库中，那么 `db.create_all()` 不会重新创建或者更新这个表。如果修改模型后要把改动应用到现有的数据库中，这一特性会带来不便。更新现有数据库表的粗暴方式是先删除旧表再重新创建：

```
>>> db.drop_all()
>>> db.create_all()
```

遗憾的是，这个方法有个我们不想看到的副作用，它把数据库中原有的数据都销毁了。本章末尾将会介绍一种更好的方式用于更新数据库。

5.8.2 插入行

下面这段代码创建了一些角色和用户：

```
>>> from hello import Role, User
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
>>> user_john = User(username='john', role=admin_role)
>>> user_susan = User(username='susan', role=user_role)
>>> user_david = User(username='david', role=user_role)
```

模型的构造函数接受的参数是使用关键字参数指定的模型属性初始值。注意，`role` 属性也可使用，虽然它不是真正的数据库列，但却是一对多关系的高级表示。这些新建对象的 `id` 属性并没有明确设定，因为主键是由 Flask-SQLAlchemy 管理的。现在这些对象只存在于 Python 中，还未写入数据库。因此 `id` 尚未赋值：

```
>>> print(admin_role.id)
None
>>> print(mod_role.id)
None
>>> print(user_role.id)
None
```

通过数据库会话管理对数据库所做的改动，在 Flask-SQLAlchemy 中，会话由 `db.session` 表示。准备把对象写入数据库之前，先要将其添加到会话中：

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role)
>>> db.session.add(user_role)
>>> db.session.add(user_john)
>>> db.session.add(user_susan)
>>> db.session.add(user_david)
```

或者简写成：

```
>>> db.session.add_all([admin_role, mod_role, user_role,
...     user_john, user_susan, user_david])
```

为了把对象写入数据库，我们要调用 `commit()` 方法提交会话：

```
>>> db.session.commit()
```

再次查看 `id` 属性，现在它们已经赋值了：

```
>>> print(admin_role.id)
1
>>> print(mod_role.id)
2
>>> print(user_role.id)
3
```

> 数据库会话 db.session 和第 4 章介绍的 Flask session 对象没有关系。数据库会话也称为事务。

数据库会话能保证数据库的一致性。提交操作使用原子方式把会话中的对象全部写入数据库。如果在写入会话的过程中发生了错误，整个会话都会失效。如果你始终把相关改动放在会话中提交，就能避免因部分更新导致的数据库不一致性。

> 数据库会话也可回滚。调用 db.session.rollback() 后，添加到数据库会话中的所有对象都会还原到它们在数据库时的状态。

5.8.3　修改行

在数据库会话上调用 add() 方法也能更新模型。我们继续在之前的 shell 会话中进行操作，下面这个例子把 "Admin" 角色重命名为 "Administrator"：

```
>>> admin_role.name = 'Administrator'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

5.8.4　删除行

数据库会话还有个 delete() 方法。下面这个例子把 "Moderator" 角色从数据库中删除：

```
>>> db.session.delete(mod_role)
>>> db.session.commit()
```

注意，删除与插入和更新一样，提交数据库会话后才会执行。

5.8.5　查询行

Flask-SQLAlchemy 为每个模型类都提供了 query 对象。最基本的模型查询是取回对应表中的所有记录：

```
>>> Role.query.all()
[<Role u'Administrator'>, <Role u'User'>]
>>> User.query.all()
[<User u'john'>, <User u'susan'>, <User u'david'>]
```

使用过滤器可以配置 `query` 对象进行更精确的数据库查询。下面这个例子查找角色为 `"User"` 的所有用户：

```
>>> User.query.filter_by(role=user_role).all()
[<User u'susan'>, <User u'david'>]
```

若要查看 SQLAlchemy 为查询生成的原生 SQL 查询语句，只需把 `query` 对象转换成字符串：

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username,
users.role_id AS users_role_id FROM users WHERE :param_1 = users.role_id'
```

如果你退出了 shell 会话，前面这些例子中创建的对象就不会以 Python 对象的形式存在，而是作为各自数据库表中的行。如果你打开了一个新的 shell 会话，就要从数据库中读取行，再重新创建 Python 对象。下面这个例子发起了一个查询，加载名为 `"User"` 的用户角色：

```
>>> user_role = Role.query.filter_by(name='User').first()
```

`filter_by()` 等过滤器在 `query` 对象上调用，返回一个更精确的 `query` 对象。多个过滤器可以一起调用，直到获得所需结果。

表 5-5 列出了可在 `query` 对象上调用的常用过滤器。完整的列表参见 SQLAlchemy 文档（http://docs.sqlalchemy.org）。

表5-5　常用的SQLAlchemy查询过滤器

过滤器	说　　明
`filter()`	把过滤器添加到原查询上，返回一个新查询
`filter_by()`	把等值过滤器添加到原查询上，返回一个新查询
`limit()`	使用指定的值限制原查询返回的结果数量，返回一个新查询
`offset()`	偏移原查询返回的结果，返回一个新查询
`order_by()`	根据指定条件对原查询结果进行排序，返回一个新查询
`group_by()`	根据指定条件对原查询结果进行分组，返回一个新查询

在查询上应用指定的过滤器后，通过调用 `all()` 执行查询，以列表的形式返回结果。除了 `all()` 之外，还有其他方法能触发查询执行。表 5-6 列出了执行查询的其他方法。

表5-6　最常使用的SQLAlchemy查询执行函数

方　法	说　　明
all()	以列表形式返回查询的所有结果
first()	返回查询的第一个结果，如果没有结果，则返回 None
first_or_404()	返回查询的第一个结果，如果没有结果，则终止请求，返回 404 错误响应
get()	返回指定主键对应的行，如果没有对应的行，则返回 None
get_or_404()	返回指定主键对应的行，如果没找到指定的主键，则终止请求，返回 404 错误响应
count()	返回查询结果的数量
paginate()	返回一个 Paginate 对象，它包含指定范围内的结果

关系和查询的处理方式类似。下面这个例子分别从关系的两端查询角色和用户之间的一对多关系：

```
>>> users = user_role.users
>>> users
[<User u'susan'>, <User u'david'>]
>>> users[0].role
<Role u'User'>
```

这个例子中的 user_role.users 查询有个小问题。执行 user_role.users 表达式时，隐含的查询会调用 all() 返回一个用户列表。query 对象是隐藏的，因此无法指定更精确的查询过滤器。就这个特定示例而言，返回一个按照字母顺序排序的用户列表可能更好。在示例 5-4 中，我们修改了关系的设置，加入了 lazy = 'dynamic' 参数，从而禁止自动执行查询。

示例 5-4　hello.py：动态关系

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role', lazy='dynamic')
    # ...
```

这样配置关系之后，user_role.users 会返回一个尚未执行的查询，因此可以在其上添加过滤器：

```
>>> user_role.users.order_by(User.username).all()
[<User u'david'>, <User u'susan'>]
>>> user_role.users.count()
2
```

5.9　在视图函数中操作数据库

前一节介绍的数据库操作可以直接在视图函数中进行。示例 5-5 展示了首页路由的新版本，已经把用户输入的名字写入了数据库。

示例 5-5　hello.py：在视图函数中操作数据库

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username = form.name.data)
            db.session.add(user)
            session['known'] = False
        else:
            session['known'] = True
        session['name'] = form.name.data
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html',
        form = form, name = session.get('name'),
        known = session.get('known', False))
```

在这个修改后的版本中，提交表单后，程序会使用 `filter_by()` 查询过滤器在数据库中查找提交的名字。变量 `known` 被写入用户会话中，因此重定向之后，可以把数据传给模板，用来显示自定义的欢迎消息。注意，要想让程序正常运行，你必须按照前面介绍的方法，在 Python shell 中创建数据库表。

对应的模板新版本如示例 5-6 所示。这个模板使用 `known` 参数在欢迎消息中加入了第二行，从而对已知用户和新用户显示不同的内容。

示例 5-6　templates/index.html

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
    {% if not known %}
    <p>Pleased to meet you!</p>
    {% else %}
    <p>Happy to see you again!</p>
    {% endif %}
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 5b` 签出程序的这个版本。

5.10　集成Python shell

每次启动 shell 会话都要导入数据库实例和模型，这真是份枯燥的工作。为了避免一直重复导入，我们可以做些配置，让 Flask-Script 的 `shell` 命令自动导入特定的对象。

若想把对象添加到导入列表中，我们要为 `shell` 命令注册一个 `make_context` 回调函数，如示例 5-7 所示。

示例 5-7　hello.py：为 `shell` 命令添加一个上下文

```
from flask.ext.script import Shell

def make_shell_context():
    return dict(app=app, db=db, User=User, Role=Role)
manager.add_command("shell", Shell(make_context=make_shell_context))
```

`make_shell_context()` 函数注册了程序、数据库实例以及模型，因此这些对象能直接导入 shell：

```
$ python hello.py shell
>>> app
<Flask 'app'>
>>> db
<SQLAlchemy engine='sqlite:////home/flask/flasky/data.sqlite'>
>>> User
<class 'app.User'>
```

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 5c` 签出程序的这个版本。

5.11　使用Flask-Migrate实现数据库迁移

在开发程序的过程中，你会发现有时需要修改数据库模型，而且修改之后还需要更新数据库。

仅当数据库表不存在时，Flask-SQLAlchemy 才会根据模型进行创建。因此，更新表的唯一方式就是先删除旧表，不过这样做会丢失数据库中的所有数据。

更新表的更好方法是使用*数据库迁移*框架。源码版本控制工具可以跟踪源码文件的变化，类似地，数据库迁移框架能跟踪数据库*模式*的变化，然后增量式的把变化应用到数据库中。

SQLAlchemy 的主力开发人员编写了一个迁移框架，称为 Alembic（https://alembic.readthedocs.org/en/latest/index.html）。除了直接使用 Alembic 之外，Flask 程序还可使用 Flask-Migrate（http://flask-migrate.readthedocs.org/en/latest/）扩展。这个扩展对 Alembic 做了轻量级包装，并集成到 Flask-Script 中，所有操作都通过 Flask-Script 命令完成。

5.11.1　创建迁移仓库

首先，我们要在虚拟环境中安装 Flask-Migrate：

```
(venv) $ pip install flask-migrate
```

这个扩展的初始化方法如示例 5-8 所示。

示例 5-8　hello.py：配置 Flask-Migrate

```
from flask.ext.migrate import Migrate, MigrateCommand

# ...

migrate = Migrate(app, db)
manager.add_command('db', MigrateCommand)
```

为了导出数据库迁移命令，Flask-Migrate 提供了一个 `MigrateCommand` 类，可附加到 Flask-Script 的 `manager` 对象上。在这个例子中，`MigrateCommand` 类使用 `db` 命令附加。

在维护数据库迁移之前，要使用 `init` 子命令创建迁移仓库：

```
(venv) $ python hello.py db init
  Creating directory /home/flask/flasky/migrations...done
  Creating directory /home/flask/flasky/migrations/versions...done
  Generating /home/flask/flasky/migrations/alembic.ini...done
  Generating /home/flask/flasky/migrations/env.py...done
  Generating /home/flask/flasky/migrations/env.pyc...done
  Generating /home/flask/flasky/migrations/README...done
  Generating /home/flask/flasky/migrations/script.py.mako...done
  Please edit configuration/connection/logging settings in
  '/home/flask/flasky/migrations/alembic.ini' before proceeding.
```

这个命令会创建 migrations 文件夹，所有迁移脚本都存放其中。

> 数据库迁移仓库中的文件要和程序的其他文件一起纳入版本控制。

5.11.2　创建迁移脚本

在 Alembic 中，数据库迁移用*迁移脚本*表示。脚本中有两个函数，分别是 `upgrade()` 和 `downgrade()`。`upgrade()` 函数把迁移中的改动应用到数据库中，`downgrade()` 函数则将改动删除。Alembic 具有添加和删除改动的能力，因此数据库可重设到修改历史的任意一点。

我们可以使用 `revision` 命令手动创建 Alembic 迁移，也可使用 `migrate` 命令自动创建。手动创建的迁移只是一个骨架，`upgrade()` 和 `downgrade()` 函数都是空的，开发者要使用

Alembic 提供的 Operations 对象指令实现具体操作。自动创建的迁移会根据模型定义和数据库当前状态之间的差异生成 upgrade() 和 downgrade() 函数的内容。

> 自动创建的迁移不一定总是正确的，有可能会漏掉一些细节。自动生成迁移脚本后一定要进行检查。

migrate 子命令用来自动创建迁移脚本：

```
(venv) $ python hello.py db migrate -m "initial migration"
INFO  [alembic.migration] Context impl SQLiteImpl.
INFO  [alembic.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate] Detected added table 'roles'
INFO  [alembic.autogenerate] Detected added table 'users'
INFO  [alembic.autogenerate.compare] Detected added index
'ix_users_username' on '['username']'
  Generating /home/flask/flasky/migrations/versions/1bc
  594146bb5_initial_migration.py...done
```

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 git checkout 5d 签出程序的这个版本。注意，你不用再生成程序的迁移，因为这个仓库已经包含了所有的迁移脚本。

5.11.3　更新数据库

检查并修正好迁移脚本之后，我们可以使用 db upgrade 命令把迁移应用到数据库中：

```
(venv) $ python hello.py db upgrade
INFO  [alembic.migration] Context impl SQLiteImpl.
INFO  [alembic.migration] Will assume non-transactional DDL.
INFO  [alembic.migration] Running upgrade None -> 1bc594146bb5, initial migration
```

对第一个迁移来说，其作用和调用 db.create_all() 方法一样。但在后续的迁移中，upgrade 命令能把改动应用到数据库中，且不影响其中保存的数据。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，请删除数据库文件 data.sqlite，然后执行 Flask-Migrate 提供的 upgrade 命令，使用这个迁移框架重新生成数据库。

数据库的设计和使用是很重要的话题，甚至有整本的书对其进行介绍。你应该把本章视作一个概览，更高级的话题会在后续各章中讨论。下一章将重点介绍电子邮件的发送。

第 6 章

电子邮件

很多类型的应用程序都需要在特定事件发生时提醒用户，而常用的通信方法是电子邮件。虽然 Python 标准库中的 smtplib 包可用在 Flask 程序中发送电子邮件，但包装了 smtplib 的 Flask-Mail 扩展能更好地和 Flask 集成。

使用Flask-Mail提供电子邮件支持

使用 `pip` 安装 Flask-Mail：

```
(venv) $ pip install flask-mail
```

Flask-Mail 连接到简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）服务器，并把邮件交给这个服务器发送。如果不进行配置，Flask-Mail 会连接 *localhost* 上的端口 25，无需验证即可发送电子邮件。表 6-1 列出了可用来设置 SMTP 服务器的配置。

表6-1　Flask-Mail SMTP服务器的配置

配　　置	默认值	说　　明
`MAIL_SERVER`	*localhost*	电子邮件服务器的主机名或 IP 地址
`MAIL_PORT`	25	电子邮件服务器的端口
`MAIL_USE_TLS`	`False`	启用传输层安全（Transport Layer Security，TLS）协议
`MAIL_USE_SSL`	`False`	启用安全套接层（Secure Sockets Layer，SSL）协议
`MAIL_USERNAME`	`None`	邮件账户的用户名
`MAIL_PASSWORD`	`None`	邮件账户的密码

在开发过程中，如果连接到外部 SMTP 服务器，则可能更方便。举个例子，示例 6-1 展示了如何配置程序，以便使用 Google Gmail 账户发送电子邮件。

示例 6-1　hello.py：配置 Flask-Mail 使用 Gmail

```
import os
# ...
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')
```

千万不要把账户密令直接写入脚本，特别是当你计划开源自己的作品时。为了保护账户信息，你需要让脚本从环境中导入敏感信息。

Flask-Mail 的初始化方法如示例 6-2 所示。

示例 6-2　hello.py：初始化 Flask-Mail

```
from flask.ext.mail import Mail
mail = Mail(app)
```

保存电子邮件服务器用户名和密码的两个环境变量要在环境中定义。如果你在 Linux 或 Mac OS X 中使用 bash，那么可以按照下面的方式设定这两个变量：

```
(venv) $ export MAIL_USERNAME=<Gmail username>
(venv) $ export MAIL_PASSWORD=<Gmail password>
```

微软 Windows 用户可按照下面的方式设定环境变量：

```
(venv) $ set MAIL_USERNAME=<Gmail username>
(venv) $ set MAIL_PASSWORD=<Gmail password>
```

在Python shell中发送电子邮件

你可以打开一个 shell 会话，发送一封测试邮件，以检查配置是否正确：

```
(venv) $ python hello.py shell
>>> from flask.ext.mail import Message
>>> from hello import mail
>>> msg = Message('test subject', sender='you@example.com',
...     recipients=['you@example.com'])
>>> msg.body = 'text body'
>>> msg.html = '<b>HTML</b> body'
>>> with app.app_context():
...    mail.send(msg)
...
```

注意，Flask-Mail 中的 `send()` 函数使用 `current_app`，因此要在激活的程序上下文中执行。

在程序中集成发送电子邮件功能

为了避免每次都手动编写电子邮件消息，我们最好把程序发送电子邮件的通用部分抽象出来，定义成一个函数。这么做还有个好处，即该函数可以使用 Jinja2 模板渲染邮件正文，灵活性极高。具体实现如示例 6-3 所示。

示例 6-3 hello.py：电子邮件支持

```
from flask.ext.mail import Message

app.config['FLASKY_MAIL_SUBJECT_PREFIX'] = '[Flasky]'
app.config['FLASKY_MAIL_SENDER'] = 'Flasky Admin <flasky@example.com>'

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    mail.send(msg)
```

这个函数用到了两个程序特定配置项，分别定义邮件主题的前缀和发件人的地址。`send_email` 函数的参数分别为收件人地址、主题、渲染邮件正文的模板和关键字参数列表。指定模板时不能包含扩展名，这样才能使用两个模板分别渲染纯文本正文和富文本正文。调用者将关键字参数传给 `render_template()` 函数，以便在模板中使用，进而生成电子邮件正文。

`index()` 视图函数很容易被扩展，这样每当表单接收新名字时，程序都会给管理员发送一封电子邮件。修改方法如示例 6-4 所示。

示例 6-4 hello.py：电子邮件示例

```
# ...
app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN')
# ...
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username=form.name.data)
            db.session.add(user)
            session['known'] = False
            if app.config['FLASKY_ADMIN']:
                send_email(app.config['FLASKY_ADMIN'], 'New User',
                           'mail/new_user', user=user)
        else:
            session['known'] = True
        session['name'] = form.name.data
```

```
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'),
                           known=session.get('known', False))
```

电子邮件的收件人保存在环境变量 FLASKY_ADMIN 中，在程序启动过程中，它会加载到一个同名配置变量中。我们要创建两个模板文件，分别用于渲染纯文本和 HTML 版本的邮件正文。这两个模板文件都保存在 templates 文件夹下的 mail 子文件夹中，以便和普通模板区分开来。电子邮件的模板中要有一个模板参数是用户，因此调用 send_mail() 函数时要以关键字参数的形式传入用户。



如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 git checkout 6a 签出程序的这个版本。

除了前面提到的环境变量 MAIL_USERNAME 和 MAIL_PASSWORD 之外，这个版本的程序还需要使用环境变量 FLASKY_ADMIN。Linux 和 Mac OS X 用户可使用下面的命令添加：

```
(venv) $ export FLASKY_ADMIN=<your-email-address>
```

对微软 Windows 用户来说，等价的命令是：

```
(venv) $ set FLASKY_ADMIN=<Gmail username>
```

设置好这些环境变量后，我们就可以测试程序了。每次你在表单中填写新名字时，管理员都会收到一封电子邮件。

异步发送电子邮件

如果你发送了几封测试邮件，可能会注意到 mail.send() 函数在发送电子邮件时停滞了几秒钟，在这个过程中浏览器就像无响应一样。为了避免处理请求过程中不必要的延迟，我们可以把发送电子邮件的函数移到后台线程中。修改方法如示例 6-5 所示。

示例 6-5　hello.py：异步发送电子邮件

```
from threading import Thread

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
```

```
    msg.html = render_template(template + '.html', **kwargs)
    thr = Thread(target=send_async_email, args=[app, msg])
    thr.start()
    return thr
```

上述实现涉及一个有趣的问题。很多 Flask 扩展都假设已经存在激活的程序上下文和请求上下文。Flask-Mail 中的 `send()` 函数使用 `current_app`，因此必须激活程序上下文。不过，在不同线程中执行 `mail.send()` 函数时，程序上下文要使用 `app.app_context()` 人工创建。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，可以执行 `git checkout 6b` 签出程序的这个版本。

现在再运行程序，你会发现程序流畅多了。不过要记住，程序要发送大量电子邮件时，使用专门发送电子邮件的作业要比给每封邮件都新建一个线程更合适。例如，我们可以把执行 `send_async_email()` 函数的操作发给 Celery（http://www.celeryproject.org/）任务队列。

至此，我们完成了对大多数 Web 程序所需功能的概述。现在的问题是，hello.py 脚本变得越来越大，难以使用。在下一章中，你会学到如何组织大型程序的结构。

第 7 章

大型程序的结构

尽管在单一脚本中编写小型 Web 程序很方便，但这种方法并不能广泛使用。程序变复杂后，使用单个大型源码文件会导致很多问题。

不同于大多数其他的 Web 框架，Flask 并不强制要求大型项目使用特定的组织方式，程序结构的组织方式完全由开发者决定。在本章，我们将介绍一种使用包和模块组织大型程序的方式。本书后续示例都将采用这种结构。

7.1 项目结构

Flask 程序的基本结构如示例 7-1 所示。

示例 7-1　多文件 Flask 程序的基本结构

```
|-flasky
  |-app/
    |-templates/
    |-static/
    |-main/
      |-__init__.py
      |-errors.py
      |-forms.py
      |-views.py
    |-__init__.py
    |-email.py
    |-models.py
  |-migrations/
  |-tests/
    |-__init__.py
```

```
    |-test*.py
  |-venv/
  |-requirements.txt
  |-config.py
  |-manage.py
```

这种结构有 4 个顶级文件夹：

- Flask 程序一般都保存在名为 app 的包中；
- 和之前一样，migrations 文件夹包含数据库迁移脚本；
- 单元测试编写在 tests 包中；
- 和之前一样，venv 文件夹包含 Python 虚拟环境。

同时还创建了一些新文件：

- requirements.txt 列出了所有依赖包，便于在其他电脑中重新生成相同的虚拟环境；
- config.py 存储配置；
- manage.py 用于启动程序以及其他的程序任务。

为了帮助你完全理解这个结构，下面几节讲解把 hello.py 程序转换成这种结构的过程。

7.2 配置选项

程序经常需要设定多个配置。这方面最好的例子就是开发、测试和生产环境要使用不同的数据库，这样才不会彼此影响。

我们不再使用 hello.py 中简单的字典状结构配置，而使用层次结构的配置类。config.py 文件的内容如示例 7-2 所示。

示例 7-2 config.py：程序的配置

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'hard to guess string'
    SQLALCHEMY_COMMIT_ON_TEARDOWN = True
    FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]'
    FLASKY_MAIL_SENDER = 'Flasky Admin <flasky@example.com>'
    FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN')

    @staticmethod
    def init_app(app):
        pass

class DevelopmentConfig(Config):
    DEBUG = True
    MAIL_SERVER = 'smtp.googlemail.com'
```

```
    MAIL_PORT = 587
    MAIL_USE_TLS = True
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')

class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data-test.sqlite')

class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data.sqlite')

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,

    'default': DevelopmentConfig
}
```

基类 Config 中包含通用配置，子类分别定义专用的配置。如果需要，你还可添加其他配置类。

为了让配置方式更灵活且更安全，某些配置可以从环境变量中导入。例如，SECRET_KEY 的值，这是个敏感信息，可以在环境中设定，但系统也提供了一个默认值，以防环境中没有定义。

在 3 个子类中，SQLALCHEMY_DATABASE_URI 变量都被指定了不同的值。这样程序就可在不同的配置环境中运行，每个环境都使用不同的数据库。

配置类可以定义 init_app() 类方法，其参数是程序实例。在这个方法中，可以执行对当前环境的配置初始化。现在，基类 Config 中的 init_app() 方法为空。

在这个配置脚本末尾，config 字典中注册了不同的配置环境，而且还注册了一个默认配置（本例的开发环境）。

7.3　程序包

程序包用来保存程序的所有代码、模板和静态文件。我们可以把这个包直接称为 app（应用），如果有需求，也可使用一个程序专用名字。templates 和 static 文件夹是程序包的一部分，因此这两个文件夹被移到了 app 中。数据库模型和电子邮件支持函数也被移到了这个包中，分别保存为 app/models.py 和 app/email.py。

7.3.1 使用程序工厂函数

在单个文件中开发程序很方便，但却有个很大的缺点，因为程序在全局作用域中创建，所以无法动态修改配置。运行脚本时，程序实例已经创建，再修改配置为时已晚。这一点对单元测试尤其重要，因为有时为了提高测试覆盖度，必须在不同的配置环境中运行程序。

这个问题的解决方法是延迟创建程序实例，把创建过程移到可显式调用的*工厂函数*中。这种方法不仅可以给脚本留出配置程序的时间，还能够创建多个程序实例，这些实例有时在测试中非常有用。程序的工厂函数在 app 包的构造文件中定义，如示例 7-3 所示。

构造文件导入了大多数正在使用的 Flask 扩展。由于尚未初始化所需的程序实例，所以没有初始化扩展，创建扩展类时没有向构造函数传入参数。`create_app()` 函数就是程序的工厂函数，接受一个参数，是程序使用的配置名。配置类在 config.py 文件中定义，其中保存的配置可以使用 Flask `app.config` 配置对象提供的 `from_object()` 方法直接导入程序。至于配置对象，则可以通过名字从 `config` 字典中选择。程序创建并配置好后，就能初始化扩展了。在之前创建的扩展对象上调用 `init_app()` 可以完成初始化过程。

示例 7-3　app/__init__.py：程序包的构造文件

```
from flask import Flask, render_template
from flask.ext.bootstrap import Bootstrap
from flask.ext.mail import Mail
from flask.ext.moment import Moment
from flask.ext.sqlalchemy import SQLAlchemy
from config import config

bootstrap = Bootstrap()
mail = Mail()
moment = Moment()
db = SQLAlchemy()

def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)

    bootstrap.init_app(app)
    mail.init_app(app)
    moment.init_app(app)
    db.init_app(app)

    # 附加路由和自定义的错误页面

    return app
```

工厂函数返回创建的程序示例，不过要注意，现在工厂函数创建的程序还不完整，因为没有路由和自定义的错误页面处理程序。这是下一节要讲的话题。

7.3.2 在蓝本中实现程序功能

转换成程序工厂函数的操作让定义路由变复杂了。在单脚本程序中，程序实例存在于全局作用域中，路由可以直接使用 `app.route` 修饰器定义。但现在程序在运行时创建，只有调用 `create_app()` 之后才能使用 `app.route` 修饰器，这时定义路由就太晚了。和路由一样，自定义的错误页面处理程序也面临相同的困难，因为错误页面处理程序使用 `app.errorhandler` 修饰器定义。

幸好 Flask 使用蓝本提供了更好的解决方法。蓝本和程序类似，也可以定义路由。不同的是，在蓝本中定义的路由处于休眠状态，直到蓝本注册到程序上后，路由才真正成为程序的一部分。使用位于全局作用域中的蓝本时，定义路由的方法几乎和单脚本程序一样。

和程序一样，蓝本可以在单个文件中定义，也可使用更结构化的方式在包中的多个模块中创建。为了获得最大的灵活性，程序包中创建了一个子包，用于保存蓝本。示例 7-4 是这个子包的构造文件，蓝本就创建于此。

示例 7-4 app/main/__init__.py：创建蓝本

```
from flask import Blueprint

main = Blueprint('main', __name__)

from . import views, errors
```

通过实例化一个 `Blueprint` 类对象可以创建蓝本。这个构造函数有两个必须指定的参数：蓝本的名字和蓝本所在的包或模块。和程序一样，大多数情况下第二个参数使用 Python 的 `__name__` 变量即可。

程序的路由保存在包里的 app/main/views.py 模块中，而错误处理程序保存在 app/main/errors.py 模块中。导入这两个模块就能把路由和错误处理程序与蓝本关联起来。注意，这些模块在 app/main/__init__.py 脚本的末尾导入，这是为了避免循环导入依赖，因为在 views.py 和 errors.py 中还要导入蓝本 `main`。

蓝本在工厂函数 `create_app()` 中注册到程序上，如示例 7-5 所示。

示例 7-5 app/_init_.py：注册蓝本

```
def create_app(config_name):
    # ...
    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    return app
```

示例 7-6 显示了错误处理程序。

示例 7-6　app/main/errors.py：蓝本中的错误处理程序

```
from flask import render_template
from . import main

@main.app_errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@main.app_errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

在蓝本中编写错误处理程序稍有不同，如果使用 `errorhandler` 修饰器，那么只有蓝本中的错误才能触发处理程序。要想注册程序全局的错误处理程序，必须使用 `app_errorhandler`。

在蓝本中定义的程序路由如示例 7-7 所示。

示例 7-7　app/main/views.py：蓝本中定义的程序路由

```
from datetime import datetime
from flask import render_template, session, redirect, url_for

from . import main
from .forms import NameForm
from .. import db
from ..models import User

@main.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('.index'))
    return render_template('index.html',
                           form=form, name=session.get('name'),
                           known=session.get('known', False),
                           current_time=datetime.utcnow())
```

在蓝本中编写视图函数主要有两点不同：第一，和前面的错误处理程序一样，路由修饰器由蓝本提供；第二，`url_for()` 函数的用法不同。你可能还记得，`url_for()` 函数的第一个参数是路由的端点名，在程序的路由中，默认为视图函数的名字。例如，在单脚本程序中，`index()` 视图函数的 URL 可使用 `url_for('index')` 获取。

在蓝本中就不一样了，Flask 会为蓝本中的全部端点加上一个命名空间，这样就可以在不同的蓝本中使用相同的端点名定义视图函数，而不会产生冲突。命名空间就是蓝本的名字（`Blueprint` 构造函数的第一个参数），所以视图函数 `index()` 注册的端点名是 `main.index`，其 URL 使用 `url_for('main.index')` 获取。

url_for() 函数还支持一种简写的端点形式，在蓝本中可以省略蓝本名，例如 url_for('.index')。在这种写法中，命名空间是当前请求所在的蓝本。这意味着同一蓝本中的重定向可以使用简写形式，但跨蓝本的重定向必须使用带有命名空间的端点名。

为了完全修改程序的页面，表单对象也要移到蓝本中，保存于 app/main/forms.py 模块。

7.4 启动脚本

顶级文件夹中的 manage.py 文件用于启动程序。脚本内容如示例 7-8 所示。

示例 7-8 manage.py：启动脚本

```
#!/usr/bin/env python
import os
from app import create_app, db
from app.models import User, Role
from flask.ext.script import Manager, Shell
from flask.ext.migrate import Migrate, MigrateCommand

app = create_app(os.getenv('FLASK_CONFIG') or 'default')
manager = Manager(app)
migrate = Migrate(app, db)

def make_shell_context():
    return dict(app=app, db=db, User=User, Role=Role)
manager.add_command("shell", Shell(make_context=make_shell_context))
manager.add_command('db', MigrateCommand)

if __name__ == '__main__':
    manager.run()
```

这个脚本先创建程序。如果已经定义了环境变量 FLASK_CONFIG，则从中读取配置名；否则使用默认配置。然后初始化 Flask-Script、Flask-Migrate 和为 Python shell 定义的上下文。

出于便利，脚本中加入了 shebang 声明，所以在基于 Unix 的操作系统中可以通过 ./manage.py 执行脚本，而不用使用复杂的 python manage.py。

7.5 需求文件

程序中必须包含一个 requirements.txt 文件，用于记录所有依赖包及其精确的版本号。如果要在另一台电脑上重新生成虚拟环境，这个文件的重要性就体现出来了，例如部署程序时使用的电脑。pip 可以使用如下命令自动生成这个文件：

```
(venv) $ pip freeze >requirements.txt
```

安装或升级包后，最好更新这个文件。需求文件的内容示例如下：

```
Flask==0.10.1
Flask-Bootstrap==3.0.3.1
Flask-Mail==0.9.0
Flask-Migrate==1.1.0
Flask-Moment==0.2.0
Flask-SQLAlchemy==1.0
Flask-Script==0.6.6
Flask-WTF==0.9.4
Jinja2==2.7.1
Mako==0.9.1
MarkupSafe==0.18
SQLAlchemy==0.8.4
WTForms==1.0.5
Werkzeug==0.9.4
alembic==0.6.2
blinker==1.3
itsdangerous==0.23
```

如果你要创建这个虚拟环境的完全副本，可以创建一个新的虚拟环境，并在其上运行以下命令：

```
(venv) $ pip install -r requirements.txt
```

当你阅读本书时，该示例 requirements.txt 文件中的版本号可能已经过期了。如果愿意，你可以试着使用这些包的最新版。如果遇到问题，你可以随时换回这个需求文件中的版本，因为这些版本和程序兼容。

7.6 单元测试

这个程序很小，所以没什么可测试的。不过为了演示，我们可以编写两个简单的测试，如示例 7-9 所示。

示例 7-9 tests/test_basics.py：单元测试

```
import unittest
from flask import current_app
from app import create_app, db

class BasicsTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_app_exists(self):
```

```
        self.assertFalse(current_app is None)

    def test_app_is_testing(self):
        self.assertTrue(current_app.config['TESTING'])
```

这个测试使用 Python 标准库中的 `unittest` 包编写。`setUp()` 和 `tearDown()` 方法分别在各测试前后运行，并且名字以 `test_` 开头的函数都作为测试执行。

> 如果你想进一步了解如何使用 Python 的 unittest 包编写测试，请阅读官方文档（https://docs.python.org/2/library/unittest.html）。

`setUp()` 方法尝试创建一个测试环境，类似于运行中的程序。首先，使用测试配置创建程序，然后激活上下文。这一步的作用是确保能在测试中使用 `current_app`，像普通请求一样。然后创建一个全新的数据库，以备不时之需。数据库和程序上下文在 `tearDown()` 方法中删除。

第一个测试确保程序实例存在。第二个测试确保程序在测试配置中运行。若想把 tests 文件夹作为包使用，需要添加 tests/__init__.py 文件，不过这个文件可以为空，因为 `unittest` 包会扫描所有模块并查找测试。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 7a` 签出程序的这个版本。为确保安装了所有依赖包，还需执行 `pip install -r requirements.txt` 命令。

为了运行单元测试，你可以在 manage.py 脚本中添加一个自定义命令。示例 7-10 展示了如何添加 `test` 命令。

示例 7-10 manage.py：启动单元测试的命令

```
@manager.command
def test():
    """Run the unit tests."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
```

`manager.command` 修饰器让自定义命令变得简单。修饰函数名就是命令名，函数的文档字符串会显示在帮助消息中。`test()` 函数的定义体中调用了 `unittest` 包提供的测试运行函数。

单元测试可使用下面的命令运行：

```
(venv) $ python manage.py test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok

.----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

7.7　创建数据库

重组后的程序和单脚本版本使用不同的数据库。

首选从环境变量中读取数据库的 URL，同时还提供了一个默认的 SQLite 数据库做备用。3 种配置环境中的环境变量名和 SQLite 数据库文件名都不一样。例如，在开发环境中，数据库 URL 从环境变量 DEV_DATABASE_URL 中读取，如果没有定义这个环境变量，则使用名为 data-dev.sqlite 的 SQLite 数据库。

不管从哪里获取数据库 URL，都要在新数据库中创建数据表。如果使用 Flask-Migrate 跟踪迁移，可使用如下命令创建数据表或者升级到最新修订版本：

```
(venv) $ python manage.py db upgrade
```

不管你是否相信，第一部分到此就要结束了。现在你已经学到了使用 Flask 开发 Web 程序的必备基础知识，不过可能还不确定如何把这些知识融贯起来开发一个真正的程序。本书第二部分的目的就是解决这个问题，带着你一步一步地开发出一个完整的程序。

第二部分

实例：社交博客程序

第 8 章

用户认证

大多数程序都要进行用户跟踪。用户连接程序时会进行身份认证，通过这一过程，让程序知道自己的身份。程序知道用户是谁后，就能提供有针对性的体验。

最常用的认证方法要求用户提供一个身份证明（用户的电子邮件或用户名）和一个密码。本章要为 Flasky 开发一个完整的认证系统。

8.1　Flask的认证扩展

优秀的 Python 认证包很多，但没有一个能实现所有功能。本章介绍的认证方案使用了多个包，并编写了胶水代码让其良好协作。本章使用的包列表如下。

- Flask-Login：管理已登录用户的用户会话。
- Werkzeug：计算密码散列值并进行核对。
- itsdangerous：生成并核对加密安全令牌。

除了认证相关的包之外，本章还用到如下常规用途的扩展。

- Flask-Mail：发送与认证相关的电子邮件。
- Flask-Bootstrap：HTML 模板。
- Flask-WTF：Web 表单。

8.2　密码安全性

设计 Web 程序时，人们往往会高估数据库中用户信息的安全性。如果攻击者入侵服务器获

取了数据库，用户的安全就处在风险之中，这个风险比你想象的要大。众所周知，大多数用户都在不同的网站中使用相同的密码，因此，即便不保存任何敏感信息，攻击者获得存储在数据库中的密码之后，也能访问用户在其他网站中的账户。

若想保证数据库中用户密码的安全，关键在于不能存储密码本身，而要存储密码的散列值。计算密码散列值的函数接收密码作为输入，使用一种或多种加密算法转换密码，最终得到一个和原始密码没有关系的字符序列。核对密码时，密码散列值可代替原始密码，因为计算散列值的函数是可复现的：只要输入一样，结果就一样。

> 计算密码散列值是个复杂的任务，很难正确处理。因此强烈建议你不要自己实现，而是使用经过社区成员审查且声誉良好的库。如果你对生成安全密码散列值的过程感兴趣，“Salted Password Hashing - Doing it Right”（计算加盐密码散列值的正确方法，https://crackstation.net/hashing-security.htm）这篇文章值得一读。

使用Werkzeug实现密码散列

Werkzeug 中的 security 模块能够很方便地实现密码散列值的计算。这一功能的实现只需要两个函数，分别用在注册用户和验证用户阶段。

- `generate_password_hash(password, method=`*`pbkdf2:sha1`*`, salt_length=8)`：这个函数将原始密码作为输入，以字符串形式输出密码的散列值，输出的值可保存在用户数据库中。`method` 和 `salt_length` 的默认值就能满足大多数需求。
- `check_password_hash(hash, password)`：这个函数的参数是从数据库中取回的密码散列值和用户输入的密码。返回值为 `True` 表明密码正确。

示例 8-1 展示了第 5 章创建的 `User` 模型为支持密码散列所做的改动。

示例 8-1 app/models.py：在 `User` 模型中加入密码散列

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    # ...
    password_hash = db.Column(db.String(128))

    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)
```

```
    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

计算密码散列值的函数通过名为 password 的只写属性实现。设定这个属性的值时，赋值方法会调用 Werkzeug 提供的 generate_password_hash() 函数，并把得到的结果赋值给 password_hash 字段。如果试图读取 password 属性的值，则会返回错误，原因很明显，因为生成散列值后就无法还原成原来的密码了。

verify_password 方法接受一个参数（即密码），将其传给 Werkzeug 提供的 check_password_hash() 函数，和存储在 User 模型中的密码散列值进行比对。如果这个方法返回 True，就表明密码是正确的。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 git checkout 8a 签出程序的这个版本。

密码散列功能已经完成，可以在 shell 中进行测试：

```
(venv) $ python manage.py shell
>>> u = User()
>>> u.password = 'cat'
>>> u.password_hash
'pbkdf2:sha1:1000$duxMk0OF$4735b293e397d6eeaf650aaf490fd9091f928bed'
>>> u.verify_password('cat')
True
>>> u.verify_password('dog')
False
>>> u2 = User()
>>> u2.password = 'cat'
>>> u2.password_hash
'pbkdf2:sha1:1000$UjvnGeTP$875e28eb0874f44101d6b332442218f66975ee89'
```

注意，即使用户 u 和 u2 使用了相同的密码，它们的密码散列值也完全不一样。为了确保这个功能今后可持续使用，我们可以把上述测试写成单元测试，以便于重复执行。我们要在 tests 包中新建一个模块，编写 3 个新测试，测试最近对 User 模型所做的修改，如示例 8-2 所示。

示例 8-2 tests/test_user_model.py：密码散列化测试

```
import unittest
from app.models import User

class UserModelTestCase(unittest.TestCase):
    def test_password_setter(self):
        u = User(password = 'cat')
        self.assertTrue(u.password_hash is not None)
```

```
    def test_no_password_getter(self):
        u = User(password = 'cat')
        with self.assertRaises(AttributeError):
            u.password

    def test_password_verification(self):
        u = User(password = 'cat')
        self.assertTrue(u.verify_password('cat'))
        self.assertFalse(u.verify_password('dog'))

    def test_password_salts_are_random(self):
        u = User(password='cat')
        u2 = User(password='cat')
        self.assertTrue(u.password_hash != u2.password_hash)
```

8.3 创建认证蓝本

我们在第 7 章介绍过蓝本，把创建程序的过程移入工厂函数后，可以使用蓝本在全局作用域中定义路由。与用户认证系统相关的路由可在 `auth` 蓝本中定义。对于不同的程序功能，我们要使用不同的蓝本，这是保持代码整齐有序的好方法。

`auth` 蓝本保存在同名 Python 包中。蓝本的包构造文件创建蓝本对象，再从 views.py 模块中引入路由，代码如示例 8-3 所示。

示例 8-3 app/auth/__init__.py：创建蓝本

```
from flask import Blueprint

auth = Blueprint('auth', __name__)

from . import views
```

app/auth/views.py 模块引入蓝本，然后使用蓝本的 `route` 修饰器定义与认证相关的路由，如示例 8-4 所示。这段代码中添加了一个 /login 路由，渲染同名占位模板。

示例 8-4 app/auth/views.py：蓝本中的路由和视图函数

```
from flask import render_template
from . import auth

@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

注意，为 `render_template()` 指定的模板文件保存在 auth 文件夹中。这个文件夹必须在 app/templates 中创建，因为 Flask 认为模板的路径是相对于程序模板文件夹而言的。为避免与 `main` 蓝本和后续添加的蓝本发生模板命名冲突，可以把蓝本使用的模板保存在单独的文件夹中。

我们也可将蓝本配置成使用其独立的文件夹保存模板。如果配置了多个模板文件夹，`render_template()` 函数会首先搜索程序配置的模板文件夹，然后再搜索蓝本配置的模板文件夹。

`auth` 蓝本要在 `create_app()` 工厂函数中附加到程序上，如示例 8-5 所示。

示例 8-5　app/__init__.py：附加蓝本

```
def create_app(config_name):
    # ...
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')

    return app
```

注册蓝本时使用的 `url_prefix` 是可选参数。如果使用了这个参数，注册后蓝本中定义的所有路由都会加上指定的前缀，即这个例子中的 /auth。例如，/login 路由会注册成 /auth/login，在开发 Web 服务器中，完整的 URL 就变成了 http://localhost:5000/auth/login。

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 8b` 签出程序的这个版本。

8.4　使用Flask-Login认证用户

用户登录程序后，他们的认证状态要被记录下来，这样浏览不同的页面时才能记住这个状态。Flask-Login 是个非常有用的小型扩展，专门用来管理用户认证系统中的认证状态，且不依赖特定的认证机制。

使用之前，我们要在虚拟环境中安装这个扩展：

```
(venv) $ pip install flask-login
```

8.4.1　准备用于登录的用户模型

要想使用 Flask-Login 扩展，程序的 `User` 模型必须实现几个方法。需要实现的方法如表 8-1 所示。

表8-1　Flask-Login要求实现的用户方法

方　　法	说　　明
is_authenticated()	如果用户已经登录，必须返回 True，否则返回 False
is_active()	如果允许用户登录，必须返回 True，否则返回 False。如果要禁用账户，可以返回 False
is_anonymous()	对普通用户必须返回 False
get_id()	必须返回用户的唯一标识符，使用 Unicode 编码字符串

这 4 个方法可以在模型类中作为方法直接实现，不过还有一种更简单的替代方案。Flask-Login 提供了一个 UserMixin 类，其中包含这些方法的默认实现，且能满足大多数需求。修改后的 User 模型如示例 8-6 所示。

示例 8-6　app/models.py：修改 User 模型，支持用户登录

```
from flask.ext.login import UserMixin

class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key = True)
    email = db.Column(db.String(64), unique=True, index=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

注意，示例中同时还添加了 email 字段。在这个程序中，用户使用电子邮件地址登录，因为相对于用户名而言，用户更不容易忘记自己的电子邮件地址。

Flask-Login 在程序的工厂函数中初始化，如示例 8-7 所示。

示例 8-7　app/__init__.py：初始化 Flask-Login

```
from flask.ext.login import LoginManager

login_manager = LoginManager()
login_manager.session_protection = 'strong'
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
```

LoginManager 对象的 session_protection 属性可以设为 None、'basic' 或 'strong'，以提供不同的安全等级防止用户会话遭篡改。设为 'strong' 时，Flask-Login 会记录客户端 IP 地址和浏览器的用户代理信息，如果发现异动就登出用户。login_view 属性设置登录页面的端点。回忆一下，登录路由在蓝本中定义，因此要在前面加上蓝本的名字。

最后，Flask-Login 要求程序实现一个回调函数，使用指定的标识符加载用户。这个函数的定义如示例 8-8 所示。

示例 8-8　app/models.py：加载用户的回调函数

```
from . import login_manager

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

加载用户的回调函数接收以 Unicode 字符串形式表示的用户标识符。如果能找到用户，这个函数必须返回用户对象；否则应该返回 None。

8.4.2　保护路由

为了保护路由只让认证用户访问，Flask-Login 提供了一个 login_required 修饰器。用法演示如下：

```
from flask.ext.login import login_required

@app.route('/secret')
@login_required
def secret():
    return 'Only authenticated users are allowed!'
```

如果未认证的用户访问这个路由，Flask-Login 会拦截请求，把用户发往登录页面。

8.4.3　添加登录表单

呈现给用户的登录表单中包含一个用于输入电子邮件地址的文本字段、一个密码字段、一个“记住我”复选框和提交按钮。这个表单使用的 Flask-WTF 类如示例 8-9 所示。

示例 8-9　app/auth/forms.py：登录表单

```
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Length, Email

class LoginForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                             Email()])
    password = PasswordField('Password', validators=[Required()])
    remember_me = BooleanField('Keep me logged in')
    submit = SubmitField('Log In')
```

电子邮件字段用到了 WTForms 提供的 Length() 和 Email() 验证函数。PasswordField 类表示属性为 type="password" 的 <input> 元素。BooleanField 类表示复选框。

登录页面使用的模板保存在 auth/login.html 文件中。这个模板只需使用 Flask-Bootstrap 提供的 wtf.quick_form() 宏渲染表单即可。登录表单在浏览器中渲染后的样子如图 8-1 所示。

base.html 模板中的导航条使用 Jinja2 条件语句，并根据当前用户的登录状态分别显示“Sign In”或“Sign Out”链接。这个条件语句如示例 8-10 所示。

示例 8-10　app/templates/base.html：导航条中的 Sign In 和 Sign Out 链接

```
<ul class="nav navbar-nav navbar-right">
    {% if current_user.is_authenticated() %}
    <li><a href="{{ url_for('auth.logout') }}">Sign Out</a></li>
    {% else %}
    <li><a href="{{ url_for('auth.login') }}">Sign In</a></li>
    {% endif %}
</ul>
```

判断条件中的变量 `current_user` 由 Flask-Login 定义，且在视图函数和模板中自动可用。这个变量的值是当前登录的用户，如果用户尚未登录，则是一个匿名用户代理对象。如果是匿名用户，`is_authenticated()` 方法返回 `False`。所以这个方法可用来判断当前用户是否已经登录。

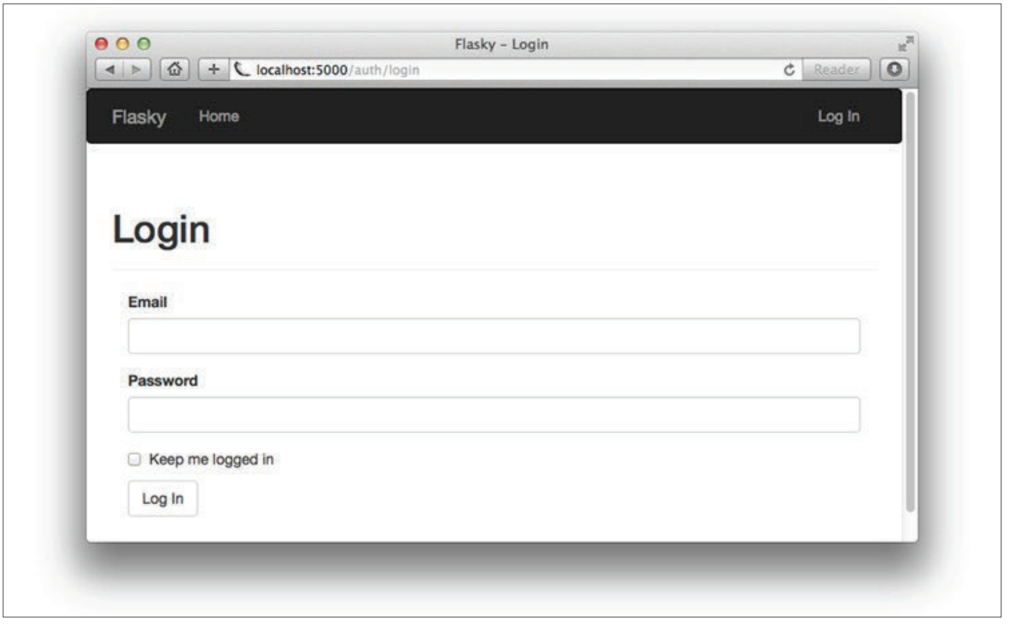


图 8-1　登录表单

8.4.4　登入用户

视图函数 `login()` 的实现如示例 8-11 所示。

示例 8-11　app/auth/views.py：登录路由

```
from flask import render_template, redirect, request, url_for, flash
from flask.ext.login import login_user
```

```python
from . import auth
from ..models import User
from .forms import LoginForm

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is not None and user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            return redirect(request.args.get('next') or url_for('main.index'))
        flash('Invalid username or password.')
    return render_template('auth/login.html', form=form)
```

这个视图函数创建了一个 `LoginForm` 对象，用法和第 4 章中的那个简单表单一样。当请求类型是 `GET` 时，视图函数直接渲染模板，即显示表单。当表单在 `POST` 请求中提交时，Flask-WTF 中的 `validate_on_submit()` 函数会验证表单数据，然后尝试登入用户。

为了登入用户，视图函数首先使用表单中填写的 `email` 从数据库中加载用户。如果电子邮件地址对应的用户存在，再调用用户对象的 `verify_password()` 方法，其参数是表单中填写的密码。如果密码正确，则调用 Flask-Login 中的 `login_user()` 函数，在用户会话中把用户标记为已登录。`login_user()` 函数的参数是要登录的用户，以及可选的"记住我"布尔值，"记住我"也在表单中填写。如果值为 `False`，那么关闭浏览器后用户会话就过期了，所以下次用户访问时要重新登录。如果值为 `True`，那么会在用户浏览器中写入一个长期有效的 cookie，使用这个 cookie 可以复现用户会话。

按照第 4 章介绍的"Post/ 重定向 /Get 模式"，提交登录密令的 `POST` 请求最后也做了重定向，不过目标 URL 有两种可能。用户访问未授权的 URL 时会显示登录表单，Flask-Login 会把原地址保存在查询字符串的 `next` 参数中，这个参数可从 `request.args` 字典中读取。如果查询字符串中没有 `next` 参数，则重定向到首页。如果用户输入的电子邮件或密码不正确，程序会设定一个 Flash 消息，再次渲染表单，让用户重试登录。

> 在生产服务器上，登录路由必须使用安全的 HTTP，从而加密传送给服务器的表单数据。如果没使用安全的 HTTP，登录密令在传输过程中可能会被截取，在服务器上花再多的精力用于保证密码安全都无济于事。

我们需要更新登录模板以渲染表单。修改内容如示例 8-12 所示。

示例 8-12 app/templates/auth/login.html：渲染登录表单

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
```

```
{% block title %}Flasky - Login{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Login</h1>
</div>
<div class="col-md-4">
    {{ wtf.quick_form(form) }}
</div>
{% endblock %}
```

8.4.5 登出用户

退出路由的实现如示例 8-13 所示。

示例 8-13 app/auth/views.py：退出路由

```
from flask.ext.login import logout_user, login_required

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    flash('You have been logged out.')
    return redirect(url_for('main.index'))
```

为了登出用户，这个视图函数调用 Flask-Login 中的 `logout_user()` 函数，删除并重设用户会话。随后会显示一个 Flash 消息，确认这次操作，再重定向到首页，这样登出就完成了。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 8c` 签出程序的这个版本。这次更新包含一个数据库迁移，所以签出代码后记得要运行 `python manage.py db upgrade`。为保证安装了所有依赖，你还要运行 `pip install -r requirements.txt`。

8.4.6 测试登录

为验证登录功能可用，可以更新首页，使用已登录用户的名字显示一个欢迎消息。模板中生成欢迎消息的部分如示例 8-14 所示。

示例 8-14 app/templates/index.html：为已登录的用户显示一个欢迎消息

```
Hello,
{% if current_user.is_authenticated() %}
    {{ current_user.username }}
{% else %}
    Stranger
{% endif %}!
```

在这个模板中再次使用 `current_user.is_authenticated()` 判断用户是否已经登录。

因为还未创建用户注册功能，所以新用户可在 shell 中注册：

```
(venv) $ python manage.py shell
>>> u = User(email='john@example.com', username='john', password='cat')
>>> db.session.add(u)
>>> db.session.commit()
```

刚刚创建的用户现在可以登录了。用户登录后显示的首页如图 8-2 所示。

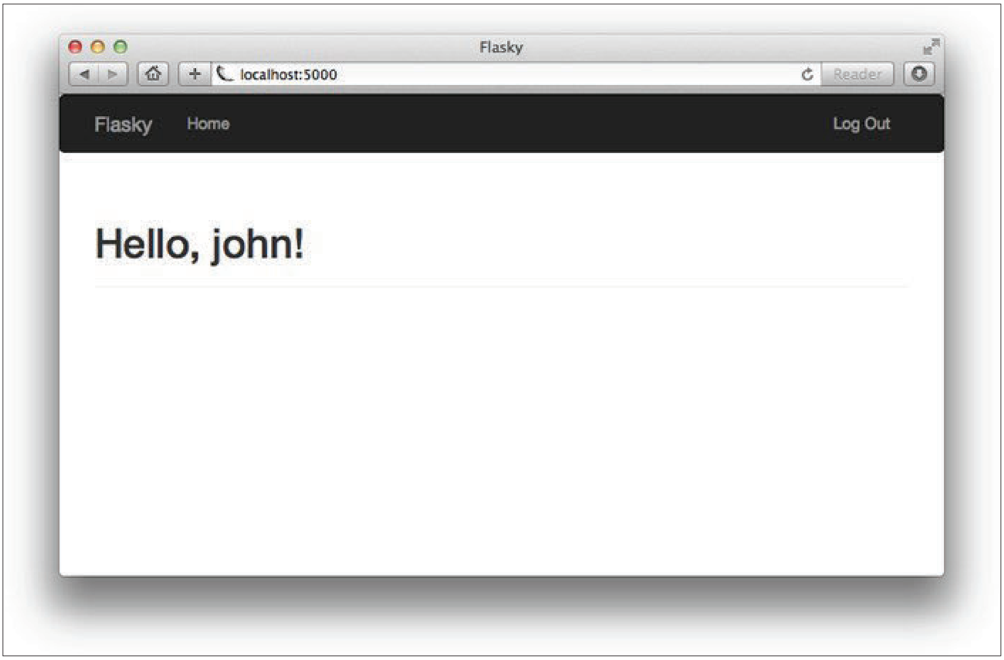


图 8-2　成功登录后的首页

8.5　注册新用户

如果新用户想成为程序的成员，必须在程序中注册，这样程序才能识别并登入用户。程序的登录页面中要显示一个链接，把用户带到注册页面，让用户输入电子邮件地址、用户名和密码。

8.5.1　添加用户注册表单

注册页面使用的表单要求用户输入电子邮件地址、用户名和密码。这个表单如示例 8-15 所示。

示例 8-15 app/auth/forms.py：用户注册表单

```
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Length, Email, Regexp, EqualTo
from wtforms import ValidationError
from ..models import User

class RegistrationForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        Required(), Length(1, 64), Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
                                          'Usernames must have only letters, '
                                          'numbers, dots or underscores')])
    password = PasswordField('Password', validators=[
        Required(), EqualTo('password2', message='Passwords must match.')])
    password2 = PasswordField('Confirm password', validators=[Required()])
    submit = SubmitField('Register')

    def validate_email(self, field):
        if User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

这个表单使用 WTForms 提供的 `Regexp` 验证函数，确保 `username` 字段只包含字母、数字、下划线和点号。这个验证函数中正则表达式后面的两个参数分别是正则表达式的旗标和验证失败时显示的错误消息。

安全起见，密码要输入两次。此时要验证两个密码字段中的值是否一致，这种验证可使用 WTForms 提供的另一验证函数实现，即 `EqualTo`。这个验证函数要附属到两个密码字段中的一个上，另一个字段则作为参数传入。

这个表单还有两个自定义的验证函数，以方法的形式实现。如果表单类中定义了以 `validate_` 开头且后面跟着字段名的方法，这个方法就和常规的验证函数一起调用。本例分别为 `email` 和 `username` 字段定义了验证函数，确保填写的值在数据库中没出现过。自定义的验证函数要想表示验证失败，可以抛出 `ValidationError` 异常，其参数就是错误消息。

显示这个表单的模板是 /templates/auth/register.html。和登录模板一样，这个模板也使用 `wtf.quick_form()` 渲染表单。注册页面如图 8-3 所示。

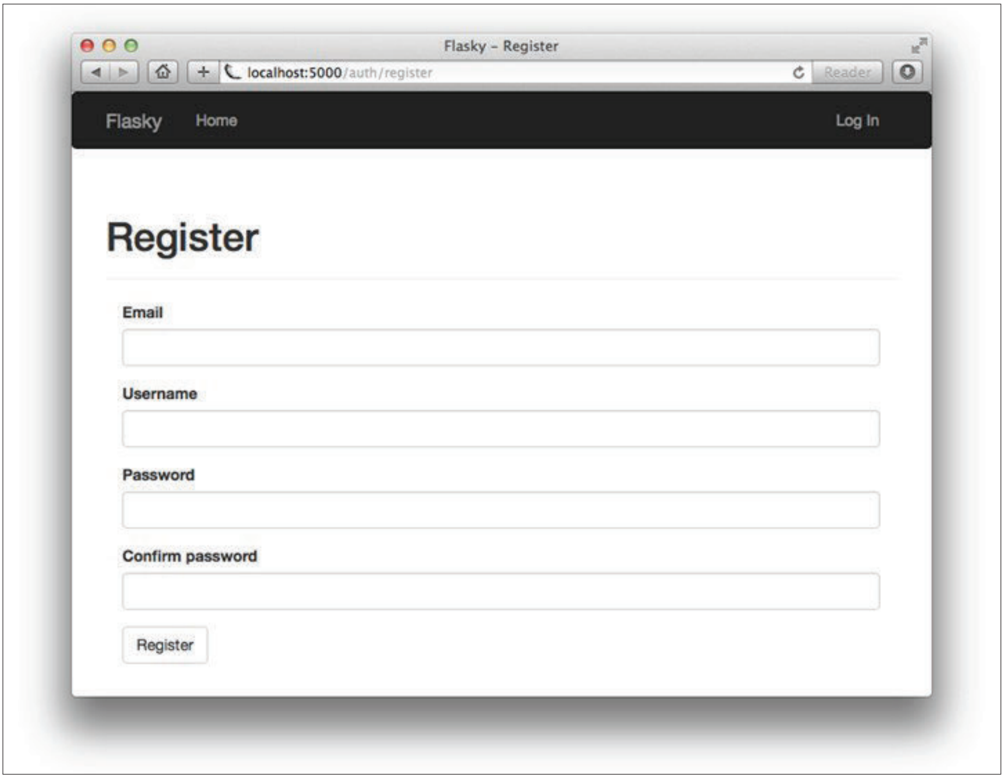


图 8-3　新用户注册表单

登录页面要显示一个指向注册页面的链接，让没有账户的用户能轻易找到注册页面。改动如示例 8-16 所示。

示例 8-16　app/templates/auth/login.html：链接到注册页面

```
<p>
    New user?
    <a href="{{ url_for('auth.register') }}">
        Click here to register
    </a>
</p>
```

8.5.2　注册新用户

处理用户注册的过程没有什么难以理解的地方。提交注册表单，通过验证后，系统就使用用户填写的信息在数据库中添加一个新用户。处理这个任务的视图函数如示例 8-17 所示。

示例 8-17　app/auth/views.py：用户注册路由

```
@auth.route('/register', methods=['GET', 'POST'])
def register():
```

```
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email=form.email.data,
                    username=form.username.data,
                    password=form.password.data)
        db.session.add(user)
        flash('You can now login.')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 8d` 签出程序的这个版本。

8.6 确认账户

对于某些特定类型的程序，有必要确认注册时用户提供的信息是否正确。常见要求是能通过提供的电子邮件地址与用户取得联系。

为验证电子邮件地址，用户注册后，程序会立即发送一封确认邮件。新账户先被标记成待确认状态，用户按照邮件中的说明操作后，才能证明自己可以被联系上。账户确认过程中，往往会要求用户点击一个包含确认令牌的特殊 URL 链接。

8.6.1 使用itsdangerous生成确认令牌

确认邮件中最简单的确认链接是 http://www.example.com/auth/confirm/<*id*> 这种形式的 URL，其中 *id* 是数据库分配给用户的数字 `id`。用户点击链接后，处理这个路由的视图函数就将收到的用户 `id` 作为参数进行确认，然后将用户状态更新为已确认。

但这种实现方式显然不是很安全，只要用户能判断确认链接的格式，就可以随便指定 URL 中的数字，从而确认任意账户。解决方法是把 URL 中的 *id* 换成将相同信息安全加密后得到的令牌。

回忆一下我们在第 4 章对用户会话的讨论，Flask 使用加密的签名 cookie 保护用户会话，防止被篡改。这种安全的 cookie 使用 itsdangerous 包签名。同样的方法也可用于确认令牌上。

下面这个简短的 shell 会话显示了如何使用 itsdangerous 包生成包含用户 `id` 的安全令牌：

```
(venv) $ python manage.py shell
>>> from manage import app
>>> from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
```

```
>>> s = Serializer(app.config['SECRET_KEY'], expires_in = 3600)
>>> token = s.dumps({ 'confirm': 23 })
>>> token
'eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.ey ...'
>>> data = s.loads(token)
>>> data
{u'confirm': 23}
```

itsdangerous 提供了多种生成令牌的方法。其中，TimedJSONWebSignatureSerializer 类生成具有过期时间的 JSON Web 签名（JSON Web Signatures，JWS）。这个类的构造函数接收的参数是一个密钥，在 Flask 程序中可使用 SECRET_KEY 设置。

dumps() 方法为指定的数据生成一个加密签名，然后再对数据和签名进行序列化，生成令牌字符串。expires_in 参数设置令牌的过期时间，单位为秒。

为了解码令牌，序列化对象提供了 loads() 方法，其唯一的参数是令牌字符串。这个方法会检验签名和过期时间，如果通过，返回原始数据。如果提供给 loads() 方法的令牌不正确或过期了，则抛出异常。

我们可以将这种生成和检验令牌的功能可添加到 User 模型中。改动如示例 8-18 所示。

示例 8-18　app/models.py：确认用户账户

```
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
from flask import current_app
from . import db

class User(UserMixin, db.Model):
    # ...
    confirmed = db.Column(db.Boolean, default=False)

    def generate_confirmation_token(self, expiration=3600):
        s = Serializer(current_app.config['SECRET_KEY'], expiration)
        return s.dumps({'confirm': self.id})

    def confirm(self, token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return False
        if data.get('confirm') != self.id:
            return False
        self.confirmed = True
        db.session.add(self)
        return True
```

generate_confirmation_token() 方法生成一个令牌，有效期默认为一小时。confirm() 方法检验令牌，如果检验通过，则把新添加的 confirmed 属性设为 True。

除了检验令牌，`confirm()` 方法还检查令牌中的 `id` 是否和存储在 `current_user` 中的已登录用户匹配。如此一来，即使恶意用户知道如何生成签名令牌，也无法确认别人的账户。

> 由于模型中新加入了一个列用来保存账户的确认状态，因此要生成并执行一个新数据库迁移。

`User` 模型中新添加的两个方法很容易进行单元测试。你可以在这个程序的 GitHub 仓库中找到单元测试。

8.6.2 发送确认邮件

当前的 /register 路由把新用户添加到数据库中后，会重定向到 /index。在重定向之前，这个路由需要发送确认邮件。改动如示例 8-19 所示。

示例 8-19 app/auth/views.py：能发送确认邮件的注册路由

```
from ..email import send_email

@auth.route('/register', methods = ['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        # ...
        db.session.add(user)
        db.session.commit()
        token = user.generate_confirmation_token()
        send_email(user.email, 'Confirm Your Account',
                   'auth/email/confirm', user=user, token=token)
        flash('A confirmation email has been sent to you by email.')
        return redirect(url_for('main.index'))
    return render_template('auth/register.html', form=form)
```

注意，即便通过配置，程序已经可以在请求末尾自动提交数据库变化，这里也要添加 `db.session.commit()` 调用。问题在于，提交数据库之后才能赋予新用户 `id` 值，而确认令牌需要用到 `id`，所以不能延后提交。

认证蓝本使用的电子邮件模板保存在 templates/auth/email 文件夹中，以便和 HTML 模板区分开来。第 6 章介绍过，一个电子邮件需要两个模板，分别用于渲染纯文本正文和富文本正文。举个例子，示例 8-20 是确认邮件模板的纯文本版本，对应的 HTML 版本可到 GitHub 仓库中查看。

示例 8-20 app/templates/auth/email/confirm.txt：确认邮件的纯文本正文

```
Dear {{ user.username }},
```

```
Welcome to Flasky!

To confirm your account please click on the following link:

{{ url_for('auth.confirm', token=token, _external=True) }}

Sincerely,

The Flasky Team

Note: replies to this email address are not monitored.
```

默认情况下，`url_for()` 生成相对 URL，例如 `url_for('auth.confirm', token='abc')` 返回的字符串是 `'/auth/confirm/abc'`。这显然不是能够在电子邮件中发送的正确 URL。相对 URL 在网页的上下文中可以正常使用，因为通过添加当前页面的主机名和端口号，浏览器会将其转换成绝对 URL。但通过电子邮件发送 URL 时，并没有这种上下文。添加到 `url_for()` 函数中的 `_external=True` 参数要求程序生成完整的 URL，其中包含协议（http:// 或 https://）、主机名和端口。

确认账户的视图函数如示例 8-21 所示。

示例 8-21　app/auth/views.py：确认用户的账户

```
from flask.ext.login import current_user

@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed:
        return redirect(url_for('main.index'))
    if current_user.confirm(token):
        flash('You have confirmed your account. Thanks!')
    else:
        flash('The confirmation link is invalid or has expired.')
    return redirect(url_for('main.index'))
```

Flask-Login 提供的 `login_required` 修饰器会保护这个路由，因此，用户点击确认邮件中的链接后，要先登录，然后才能执行这个视图函数。

这个函数先检查已登录的用户是否已经确认过，如果确认过，则重定向到首页，因为很显然此时不用做什么操作。这样处理可以避免用户不小心多次点击确认令牌带来的额外工作。

由于令牌确认完全在 `User` 模型中完成，所以视图函数只需调用 `confirm()` 方法即可，然后再根据确认结果显示不同的 Flash 消息。确认成功后，`User` 模型中 `confirmed` 属性的值会被修改并添加到会话中，请求处理完后，这两个操作被提交到数据库。

每个程序都可以决定用户确认账户之前可以做哪些操作。比如，允许未确认的用户登录，

但只显示一个页面，这个页面要求用户在获取权限之前先确认账户。

这一步可使用 Flask 提供的 before_request 钩子完成，我们在第 2 章就已经简单介绍过钩子的相关内容。对蓝本来说，before_request 钩子只能应用到属于蓝本的请求上。若想在蓝本中使用针对程序全局请求的钩子，必须使用 before_app_request 修饰器。示例 8-22 展示了如何实现这个处理程序。

示例 8-22　app/auth/views.py：在 before_app_request 处理程序中过滤未确认的账户

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated() \
            and not current_user.confirmed \
            and request.endpoint[:5] != 'auth.':
            and request.endpoint != 'static':
        return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed')
def unconfirmed():
    if current_user.is_anonymous() or current_user.confirmed:
        return redirect(url_for('main.index'))
    return render_template('auth/unconfirmed.html')
```

同时满足以下 3 个条件时，before_app_request 处理程序会拦截请求。

(1) 用户已登录（current_user.is_authenticated() 必须返回 True）。
(2) 用户的账户还未确认。
(3) 请求的端点（使用 request.endpoint 获取）不在认证蓝本中。访问认证路由要获取权限，因为这些路由的作用是让用户确认账户或执行其他账户管理操作。

如果请求满足以上 3 个条件，则会被重定向到 /auth/unconfirmed 路由，显示一个确认账户相关信息的页面。

> 如果 before_request 或 before_app_request 的回调返回响应或重定向，Flask 会直接将其发送至客户端，而不会调用请求的视图函数。因此，这些回调可在必要时拦截请求。

显示给未确认用户的页面（如图 8-4 所示）只渲染一个模板，其中有如何确认账户的说明，此外还提供了一个链接，用于请求发送新的确认邮件，以防之前的邮件丢失。重新发送确认邮件的路由如示例 8-23 所示。

示例 8-23　app/auth/views.py：重新发送账户确认邮件

```
@auth.route('/confirm')
@login_required
def resend_confirmation():
```

```
    token = current_user.generate_confirmation_token()
    send_email(current_user.email, 'Confirm Your Account',
               'auth/email/confirm', user=current_user, token=token)
    flash('A new confirmation email has been sent to you by email.')
    return redirect(url_for('main.index'))
```

这个路由为 `current_user`（即已登录的用户，也是目标用户）重做了一遍注册路由中的操作。这个路由也用 `login_required` 保护，确保访问时程序知道请求再次发送邮件的是哪个用户。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 8e` 签出程序的这个版本。这个版本包含一个数据库迁移，所以签出代码后要执行 `python manage.py db upgrade`。

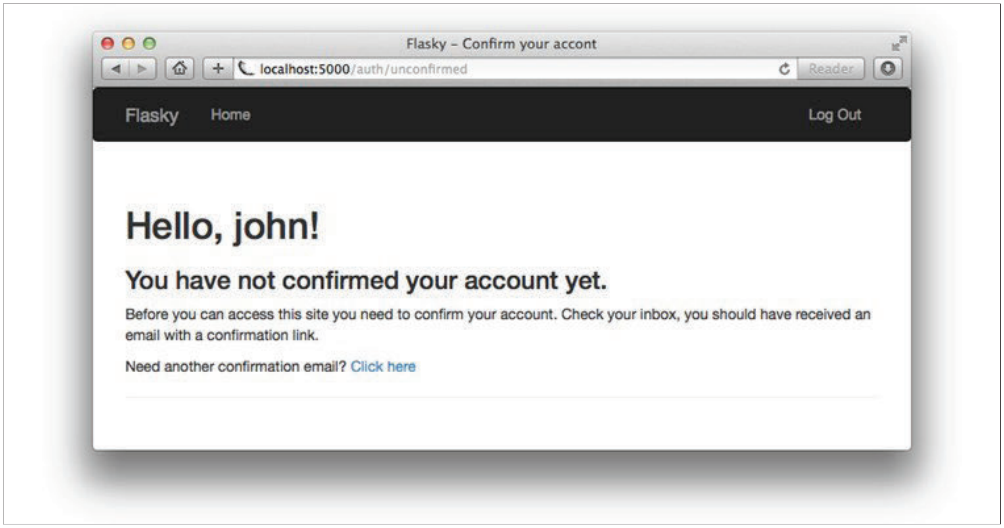


图 8-4　未确认账户页面

8.7　管理账户

拥有程序账户的用户有时可能需要修改账户信息。下面这些操作可使用本章介绍的技术添加到验证蓝本中。

修改密码

安全意识强的用户可能希望定期修改密码。这是一个很容易实现的功能，只要用户处于登录状态，就可以放心显示一个表单，要求用户输入旧密码和替换的新密码。（这个功能的实现参见 GitHub 仓库中标签为 `8f` 的提交。）

重设密码

为避免用户忘记密码无法登入的情况，程序可以提供重设密码功能。安全起见，有必要使用类似于确认账户时用到的令牌。用户请求重设密码后，程序会向用户注册时提供的电子邮件地址发送一封包含重设令牌的邮件。用户点击邮件中的链接，令牌验证后，会显示一个用于输入新密码的表单。（这个功能的实现参见 GitHub 仓库中标签为 `8g` 的提交。）

修改电子邮件地址

程序可以提供修改注册电子邮件地址的功能，不过接受新地址之前，必须使用确认邮件进行验证。使用这个功能时，用户在表单中输入新的电子邮件地址。为了验证这个地址，程序会发送一封包含令牌的邮件。服务器收到令牌后，再更新用户对象。服务器收到令牌之前，可以把新电子邮件地址保存在一个新数据库字段中作为待定地址，或者将其和 `id` 一起保存在令牌中。（这个功能的实现参见 GitHub 仓库中标签为 `8h` 的提交。）

下一章，我们使用用户角色扩充 Flasky 的用户子系统。

第 9 章

用户角色

Web 程序中的用户并非都具有同样地位。在大多数程序中，一小部分可信用户具有额外权限，用于保证程序平稳运行。管理员就是最好的例子，但有时也需要介于管理员和普通用户之间的角色，例如内容协管员。

有多种方法可用于在程序中实现角色。具体采用何种实现方法取决于所需角色的数量和细分程度。例如，简单的程序可能只需要两个角色，一个表示普通用户，一个表示管理员。对于这种情况，在 `User` 模型中添加一个 `is_administrator` 布尔值字段就足够了。复杂的程序可能需要在普通用户和管理员之间再细分出多个不同等级的角色。有些程序甚至不能使用分立的角色，这时赋予用户某些权限的组合或许更合适。

本章介绍的用户角色实现方式结合了分立的角色和权限，赋予用户分立的角色，但角色使用权限定义。

9.1　角色在数据库中的表示

第 5 章创建了一个简单的 `roles` 表，用来演示一对多关系。示例 9-1 是改进后的 `Role` 模型。

示例 9-1　app/models.py：角色的权限

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')
```

只有一个角色的 default 字段要设为 True，其他都设为 False。用户注册时，其角色会被设为默认角色。

这个模型的第二处改动是添加了 permissions 字段，其值是一个整数，表示位标志。各操作都对应一个位位置，能执行某项操作的角色，其位会被设为 1。

显然，各操作所需的程序权限是不一样的。对 Flasky 开说，各种操作如表 9-1 所示。

表9-1　程序的权限

操　　作	位　　值	说　　明
关注用户	0b00000001（0x01）	关注其他用户
在他人的文章中发表评论	0b00000010（0x02）	在他人撰写的文章中发布评论
写文章	0b00000100（0x04）	写原创文章
管理他人发表的评论	0b00001000（0x08）	查处他人发表的不当评论
管理员权限	0b10000000（0x80）	管理网站

注意，操作的权限使用 8 位表示，现在只用了其中 5 位，其他 3 位可用于将来的扩充。

表 9-1 中的权限可使用示例 9-2 中的代码表示。

示例 9-2　app/models.py：权限常量

```
class Permission:
    FOLLOW = 0x01
    COMMENT = 0x02
    WRITE_ARTICLES = 0x04
    MODERATE_COMMENTS = 0x08
    ADMINISTER = 0x80
```

表 9-2 列出了要支持的用户角色以及定义角色使用的权限位。

表9-2　用户角色

用户角色	权　　限	说　　明
匿名	0b00000000（0x00）	未登录的用户。在程序中只有阅读权限
用户	0b00000111（0x07）	具有发布文章、发表评论和关注其他用户的权限。这是新用户的默认角色
协管员	0b00001111（0x0f）	增加审查不当评论的权限
管理员	0b11111111（0xff）	具有所有权限，包括修改其他用户所属角色的权限

使用权限组织角色，这一做法让你以后添加新角色时只需使用不同的权限组合即可。

将角色手动添加到数据库中既耗时又容易出错。作为替代，我们要在 Role 类中添加一个类方法，完成这个操作，如示例 9-3 所示。

示例 9-3　app/models.py:：在数据库中创建角色

```
class Role(db.Model):
```

```
    # ...
    @staticmethod
    def insert_roles():
        roles = {
            'User': (Permission.FOLLOW |
                     Permission.COMMENT |
                     Permission.WRITE_ARTICLES, True),
            'Moderator': (Permission.FOLLOW |
                          Permission.COMMENT |
                          Permission.WRITE_ARTICLES |
                          Permission.MODERATE_COMMENTS, False),
            'Administrator': (0xff, False)
        }
        for r in roles:
            role = Role.query.filter_by(name=r).first()
            if role is None:
                role = Role(name=r)
            role.permissions = roles[r][0]
            role.default = roles[r][1]
            db.session.add(role)
        db.session.commit()
```

insert_roles() 函数并不直接创建新角色对象，而是通过角色名查找现有的角色，然后再进行更新。只有当数据库中没有某个角色名时才会创建新角色对象。如此一来，如果以后更新了角色列表，就可以执行更新操作了。要想添加新角色，或者修改角色的权限，修改 roles 数组，再运行函数即可。注意，"匿名"角色不需要在数据库中表示出来，这个角色的作用就是为了表示不在数据库中的用户。

若想把角色写入数据库，可使用 shell 会话：

```
(venv) $ python manage.py shell
>>> Role.insert_roles()
>>> Role.query.all()
[<Role u'Administrator'>, <Role u'User'>, <Role u'Moderator'>]
```

9.2　赋予角色

用户在程序中注册账户时，会被赋予适当的角色。大多数用户在注册时赋予的角色都是"用户"，因为这是默认角色。唯一的例外是管理员，管理员在最开始就应该赋予"管理员"角色。管理员由保存在设置变量 FLASKY_ADMIN 中的电子邮件地址识别，只要这个电子邮件地址出现在注册请求中，就会被赋予正确的角色。示例 9-4 展示了如何在 User 模型的构造函数中完成这一操作。

示例 9-4　app/models.py：定义默认的用户角色

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super(User, self).__init__(**kwargs)
```

```
        if self.role is None:
            if self.email == current_app.config['FLASKY_ADMIN']:
                self.role = Role.query.filter_by(permissions=0xff).first()
            if self.role is None:
                self.role = Role.query.filter_by(default=True).first()
    # ...
```

User 类的构造函数首先调用基类的构造函数，如果创建基类对象后还没定义角色，则根据电子邮件地址决定将其设为管理员还是默认角色。

9.3 角色验证

为了简化角色和权限的实现过程，我们可在 User 模型中添加一个辅助方法，检查是否有指定的权限，如示例 9-5 所示。

示例 9-5 app/models.py：检查用户是否有指定的权限

```
from flask.ext.login import UserMixin, AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...

    def can(self, permissions):
        return self.role is not None and \
            (self.role.permissions & permissions) == permissions

    def is_administrator(self):
        return self.can(Permission.ADMINISTER)

class AnonymousUser(AnonymousUserMixin):
    def can(self, permissions):
        return False

    def is_administrator(self):
        return False

login_manager.anonymous_user = AnonymousUser
```

User 模型中添加的 can() 方法在请求和赋予角色这两种权限之间进行位与操作。如果角色中包含请求的所有权限位，则返回 True，表示允许用户执行此项操作。检查管理员权限的功能经常用到，因此使用单独的方法 is_administrator() 实现。

出于一致性考虑，我们还定义了 AnonymousUser 类，并实现了 can() 方法和 is_administrator() 方法。这个对象继承自 Flask-Login 中的 AnonymousUserMixin 类，并将其设为用户未登录时 current_user 的值。这样程序不用先检查用户是否登录，就能自由调用 current_user.can() 和 current_user.is_administrator()。

如果你想让视图函数只对具有特定权限的用户开放，可以使用自定义的修饰器。示例 9-6 实现了两个修饰器，一个用来检查常规权限，一个专门用来检查管理员权限。

示例 9-6　app/decorators.py：检查用户权限的自定义修饰器

```
from functools import wraps
from flask import abort
from flask.ext.login import current_user

def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.can(permission):
                abort(403)
            return f(*args, **kwargs)
        return decorated_function
    return decorator

def admin_required(f):
    return permission_required(Permission.ADMINISTER)(f)
```

这两个修饰器都使用了 Python 标准库中的 functools 包，如果用户不具有指定权限，则返回 403 错误码，即 HTTP“禁止”错误。我们在第 3 章为 404 和 500 错误编写了自定义的错误页面，所以现在也要添加一个 403 错误页面。

下面我们举两个例子演示如何使用这些修饰器。

```
from decorators import admin_required, permission_required
from .models import Permission

@main.route('/admin')
@login_required
@admin_required
def for_admins_only():
    return "For administrators!"

@main.route('/moderator')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def for_moderators_only():
    return "For comment moderators!"
```

在模板中可能也需要检查权限，所以 `Permission` 类为所有位定义了常量以便于获取。为了避免每次调用 `render_template()` 时都多添加一个模板参数，可以使用上下文处理器。上下文处理器能让变量在所有模板中全局可访问。修改方法如示例 9-7 所示。

示例 9-7　app/main/__init__.py：把 Permission 类加入模板上下文

```
@main.app_context_processor
def inject_permissions():
    return dict(Permission=Permission)
```

新添加的角色和权限可在单元测试中进行测试。示例 9-8 是两个简单的测试，同时也演示了用法。

示例 9-8　tests/test_user_model.py：角色和权限的单元测试

```
class UserModelTestCase(unittest.TestCase):
    # ...

    def test_roles_and_permissions(self):
        Role.insert_roles()
        u = User(email='john@example.com', password='cat')
        self.assertTrue(u.can(Permission.WRITE_ARTICLES))
        self.assertFalse(u.can(Permission.MODERATE_COMMENTS))

    def test_anonymous_user(self):
        u = AnonymousUser()
        self.assertFalse(u.can(Permission.FOLLOW))
```

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 9a` 签出程序的这个版本。这个版本包含一个数据库迁移，签出代码后记得要运行 `python manage.py db upgrade`。

在你阅读下一章之前，最好重新创建或者更新开发数据库，如此一来，那些在实现角色和权限之前创建的用户账户就被赋予了角色。

现在，用户系统基本完成了。在下一章，我们要利用这个系统创建用户资料页面。

第10章

用户资料

在本章，我们要实现 Flasky 的用户资料页面。所有社交网站都会给用户提供资料页面，其中简要显示了用户在网站中的活动情况。用户可以把资料页面的 URL 分享给别人，以此宣告自己在这个网站上。因此，这个页面的 URL 要简短易记。

10.1 资料信息

为了让用户的资料页面更吸引人，我们可以在其中添加一些关于用户的其他信息。示例 10-1 扩充了 User 模型，添加了几个新字段。

示例 10-1 app/models.py：用户信息字段

```
class User(UserMixin, db.Model):
    # ...
    name = db.Column(db.String(64))
    location = db.Column(db.String(64))
    about_me = db.Column(db.Text())
    member_since = db.Column(db.DateTime(), default=datetime.utcnow)
    last_seen = db.Column(db.DateTime(), default=datetime.utcnow)
```

新添加的字段保存用户的真实姓名、所在地、自我介绍、注册日期和最后访问日期。about_me 字段的类型是 db.Text()。db.String 和 db.Text 的区别在于后者不需要指定最大长度。

两个时间戳的默认值都是当前时间。注意，datetime.utcnow 后面没有 ()，因为 db.Column() 的 default 参数可以接受函数作为默认值，所以每次需要生成默认值时，db.Column() 都会调用指定的函数。member_since 字段只需要默认值即可。

last_seen 字段创建时的初始值也是当前时间，但用户每次访问网站后，这个值都会被刷新。我们可以在 User 模型中添加一个方法完成这个操作，如示例 10-2 所示。

示例 10-2　app/models.py：刷新用户的最后访问时间

```
class User(UserMixin, db.Model):
    # ...

    def ping(self):
        self.last_seen = datetime.utcnow()
        db.session.add(self)
```

每次收到用户的请求时都要调用 ping() 方法。由于 auth 蓝本中的 before_app_request 处理程序会在每次请求前运行，所以能很轻松地实现这个需求，如示例 10-3 所示。

示例 10-3　app/auth/views.py：更新已登录用户的访问时间

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated():
        current_user.ping()
        if not current_user.confirmed \
                and request.endpoint[:5] != 'auth.':
            return redirect(url_for('auth.unconfirmed'))
```

10.2　用户资料页面

为每个用户都创建资料页面并没有什么难度。示例 10-4 显示了路由定义。

示例 10-4　app/main/views.py：资料页面的路由

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        abort(404)
    return render_template('user.html', user=user)
```

这个路由在 main 蓝本中添加。对于名为 john 的用户，其资料页面的地址是 http://localhost:5000/user/john。这个视图函数会在数据库中搜索 URL 中指定的用户名，如果找到，则渲染模板 user.html，并把用户名作为参数传入模板。如果传入路由的用户名不存在，则返回 404 错误。user.html 模板应该渲染保存在用户对象中的信息。这个模板的初始版本如示例 10-5 所示。

示例 10-5　app/templates/user.html：用户资料页面的模板

```
{% block page_content %}
<div class="page-header">
    <h1>{{ user.username }}</h1>
    {% if user.name or user.location %}
    <p>
        {% if user.name %}{{ user.name }}{% endif %}
```

```
        {% if user.location %}
            From <a href="http://maps.google.com/?q={{ user.location }}">
                {{ user.location }}
            </a>
        {% endif %}
    </p>
    {% endif %}
    {% if current_user.is_administrator() %}
    <p><a href="mailto:{{ user.email }}">{{ user.email }}</a></p>
    {% endif %}
    {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
    <p>
        Member since {{ moment(user.member_since).format('L') }}.
        Last seen {{ moment(user.last_seen).fromNow() }}.
    </p>
</div>
{% endblock %}
```

在这个模板中，有几处实现细节需要说明一下。

- `name` 和 `location` 字段在同一个 `<p>` 元素中渲染。只有至少定义了这两个字段中的一个时，`<p>` 元素才会创建。
- 用户的 `location` 字段被渲染成指向谷歌地图的查询链接。
- 如果登录用户是管理员，那么就显示用户的电子邮件地址，且渲染成 mailto 链接。

大多数用户都希望能很轻松地访问自己的资料页面，因此我们可以在导航条中添加一个链接。对 base.html 模板所做的修改如示例 10-6 所示。

示例 10-6　app/templates/base.html

```
{% if current_user.is_authenticated() %}
<li>
    <a href="{{ url_for('main.user', username=current_user.username) }}">
        Profile
    </a>
</li>
{% endif %}
```

把资料页面的链接包含在条件语句中是非常必要的，因为未认证的用户也能看到导航条，但我们不应该让他们看到资料页面的链接。

图 10-1 展示了资料页面在浏览器中的样子。图中还显示了刚添加的资料页面链接。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 10a` 签出程序的这个版本。这个版本包含一个数据库迁移，签出代码后记得运行 `python manage.py db upgrade`。

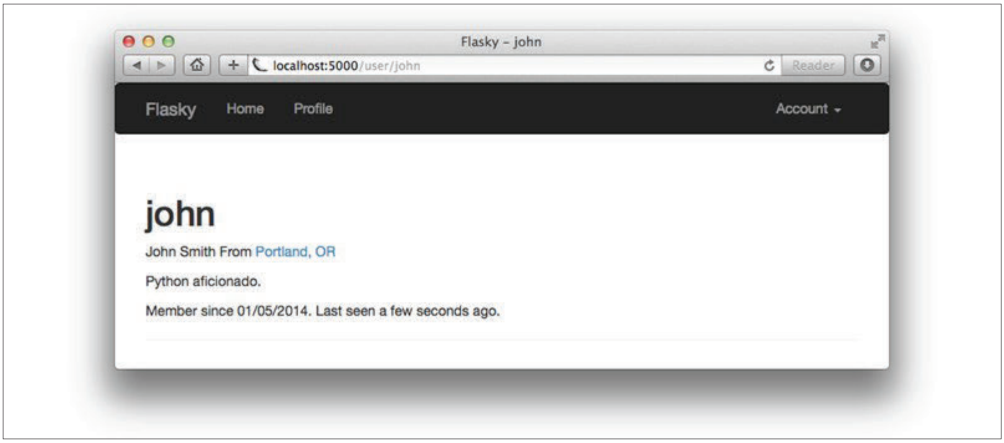


图 10-1　用户资料页面

10.3　资料编辑器

用户资料的编辑分两种情况。最显而易见的情况是，用户要进入一个页面并在其中输入自己的资料，而且这些内容显示在自己的资料页面上。还有一种不太明显但也同样重要的情况，那就是要让管理员能够编辑任意用户的资料——不仅要能编辑用户的个人信息，还要能编辑用户不能直接访问的 `User` 模型字段，例如用户角色。这两种编辑需求有本质上的区别，所以我们要创建两个不同的表单。

10.3.1　用户级别的资料编辑器

普通用户的资料编辑表单如示例 10-7 所示。

示例 10-7　app/main/forms.py：资料编辑表单

```
class EditProfileForm(Form):
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')
```

注意，这个表单中的所有字段都是可选的，因此长度验证函数允许长度为零。显示这个表单的路由定义如示例 10-8 所示。

示例 10-8　app/main/views.py：资料编辑路由

```
@main.route('/edit-profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
```

```
        current_user.name = form.name.data
        current_user.location = form.location.data
        current_user.about_me = form.about_me.data
        db.session.add(current_user)
        flash('Your profile has been updated.')
        return redirect(url_for('.user', username=current_user.username))
    form.name.data = current_user.name
    form.location.data = current_user.location
    form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', form=form)
```

在显示表单之前，这个视图函数为所有字段设定了初始值。对于所有给定字段，这一工作都是通过把初始值赋值给 form.<field-name>.data 完成的。当 form.validate_on_submit() 返回 False 时，表单中的 3 个字段都使用 current_user 中保存的初始值。提交表单后，表单字段的 data 属性中保存有更新后的值，因此可以将其赋值给用户对象中的各字段，然后再把用户对象添加到数据库会话中。编辑资料页面如图 10-2 所示。

为了让用户能轻易找到编辑页面，我们可以在资料页面中添加一个链接，如示例 10-9 所示。

示例 10-9　app/templates/user.html：资料编辑的链接

```
{% if user == current_user %}
<a class="btn btn-default" href="{{ url_for('.edit_profile') }}">
    Edit Profile
</a>
{% endif %}
```

链接外层的条件语句能确保只有当用户查看自己的资料页面时才显示这个链接。

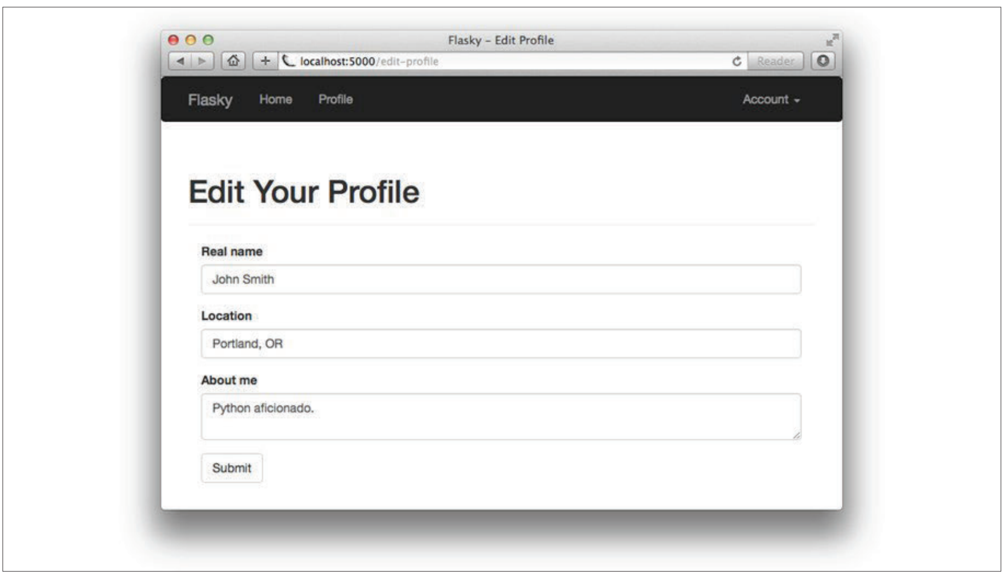


图 10-2　资料编辑器

10.3.2 管理员级别的资料编辑器

管理员使用的资料编辑表单比普通用户的表单更加复杂。除了前面的 3 个资料信息字段之外，管理员在表单中还要能编辑用户的电子邮件、用户名、确认状态和角色。这个表单如示例 10-10 所示。

示例 10-10　app/main/forms.py：管理员使用的资料编辑表单

```
class EditProfileAdminForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        Required(), Length(1, 64), Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
                                          'Usernames must have only letters, '
                                          'numbers, dots or underscores')])
    confirmed = BooleanField('Confirmed')
    role = SelectField('Role', coerce=int)
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')

    def __init__(self, user, *args, **kwargs):
        super(EditProfileAdminForm, self).__init__(*args, **kwargs)
        self.role.choices = [(role.id, role.name)
                             for role in Role.query.order_by(Role.name).all()]
        self.user = user

    def validate_email(self, field):
        if field.data != self.user.email and \
                User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if field.data != self.user.username and \
                User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

WTForms 对 HTML 表单控件 `<select>` 进行 `SelectField` 包装，从而实现下拉列表，用来在这个表单中选择用户角色。`SelectField` 实例必须在其 `choices` 属性中设置各选项。选项必须是一个由元组组成的列表，各元组都包含两个元素：选项的标识符和显示在控件中的文本字符串。`choices` 列表在表单的构造函数中设定，其值从 `Role` 模型中获取，使用一个查询按照角色名的字母顺序排列所有角色。元组中的标识符是角色的 `id`，因为这是个整数，所以在 `SelectField` 构造函数中添加 `coerce=int` 参数，从而把字段的值转换为整数，而不使用默认的字符串。

`email` 和 `username` 字段的构造方式和认证表单中的一样，但处理验证时需要更加小心。验证这两个字段时，首先要检查字段的值是否发生了变化，如果有变化，就要保证新值不和其他用户的相应字段值重复；如果字段值没有变化，则应该跳过验证。为了实现这个逻

辑，表单构造函数接收用户对象作为参数，并将其保存在成员变量中，随后自定义的验证方法要使用这个用户对象。

管理员的资料编辑器路由定义如示例 10-11 所示。

示例 10-11　app/main/views.py：管理员的资料编辑路由

```
@main.route('/edit-profile/<int:id>', methods=['GET', 'POST'])
@login_required
@admin_required
def edit_profile_admin(id):
    user = User.query.get_or_404(id)
    form = EditProfileAdminForm(user=user)
    if form.validate_on_submit():
        user.email = form.email.data
        user.username = form.username.data
        user.confirmed = form.confirmed.data
        user.role = Role.query.get(form.role.data)
        user.name = form.name.data
        user.location = form.location.data
        user.about_me = form.about_me.data
        db.session.add(user)
        flash('The profile has been updated.')
        return redirect(url_for('.user', username=user.username))
    form.email.data = user.email
    form.username.data = user.username
    form.confirmed.data = user.confirmed
    form.role.data = user.role_id
    form.name.data = user.name
    form.location.data = user.location
    form.about_me.data = user.about_me
    return render_template('edit_profile.html', form=form, user=user)
```

这个路由和较简单的、普通用户的编辑路由具有基本相同的结构。在这个视图函数中，用户由 `id` 指定，因此可使用 Flask-SQLAlchemy 提供的 `get_or_404()` 函数，如果提供的 `id` 不正确，则会返回 404 错误。

我们还需要再探讨一下用于选择用户角色的 `SelectField`。设定这个字段的初始值时，`role_id` 被赋值给了 `field.role.data`，这么做的原因在于 `choices` 属性中设置的元组列表使用数字标识符表示各选项。表单提交后，`id` 从字段的 `data` 属性中提取，并且查询时会使用提取出来的 `id` 值加载角色对象。表单中声明 `SelectField` 时使用 `coerce=int` 参数，其作用是保证这个字段的 `data` 属性值是整数。

为链接到这个页面，我们还需在用户资料页面中添加一个链接按钮，如示例 10-12 所示。

示例 10-12　app/templates/user.html：管理员使用的资料编辑链接

```
{% if current_user.is_administrator() %}
<a class="btn btn-danger"
        href="{{ url_for('.edit_profile_admin', id=user.id) }}">
    Edit Profile [Admin]
```

```
</a>
{% endif %}
```

为了醒目，这个按钮使用了不同的 Bootstrap 样式进行渲染。这里使用的条件语句确保只当登录用户为管理员时才显示按钮。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 10b` 签出程序的这个版本。

10.4 用户头像

通过显示用户的头像，我们可以进一步改进资料页面的外观。在本节，你会学到如何添加 Gravatar（http://gravatar.com/）提供的用户头像。Gravatar 是一个行业领先的头像服务，能把头像和电子邮件地址关联起来。用户先要到 http://gravatar.com 中注册账户，然后上传图片。生成头像的 URL 时，要计算电子邮件地址的 MD5 散列值：

```
(venv) $ python
>>> import hashlib
>>> hashlib.md5('john@example.com'.encode('utf-8')).hexdigest()
'd4c74594d841139328695756648b6bd6'
```

生成的头像 URL 是在 http://www.gravatar.com/avatar/ 或 https://secure.gravatar.com/avatar/ 之后加上这个 MD5 散列值。例如，你在浏览器的地址栏中输入 http://www.gravatar.com/avatar/d4c74594d841139328695756648b6bd6，就会看到电子邮件地址 `john@example.com` 对应的头像图片。如果这个电子邮件地址没有对应的头像，则会显示一个默认图片。头像 URL 的查询字符串中可以包含多个参数以配置头像图片的特征。可设参数如表 10-1 所示。

表10-1　Gravatar查询字符串参数

参数名	说　明
`s`	图片大小，单位为像素
`r`	图片级别。可选值有 `"g"`、`"pg"`、`"r"` 和 `"x"`
`d`	没有注册 Gravatar 服务的用户使用的默认图片生成方式。可选值有：`"404"`，返回 404 错误；默认图片的 URL；图片生成器 `"mm"`、`"identicon"`、`"monsterid"`、`"wavatar"`、`"retro"` 或 `"blank"` 之一
`fd`	强制使用默认头像

我们可将构建 Gravatar URL 的方法添加到 `User` 模型中，实现方式如示例 10-13 所示。

示例 10-13　app/models.py：生成 Gravatar URL

```
import hashlib
from flask import request
```

```
class User(UserMixin, db.Model):
    # ...
    def gravatar(self, size=100, default='identicon', rating='g'):
        if request.is_secure:
            url = 'https://secure.gravatar.com/avatar'
        else:
            url = 'http://www.gravatar.com/avatar'
        hash = hashlib.md5(self.email.encode('utf-8')).hexdigest()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

这一实现会选择标准的或加密的 Gravatar URL 基以匹配用户的安全需求。头像的 URL 由 URL 基、用户电子邮件地址的 MD5 散列值和参数组成，而且各参数都设定了默认值。有了上述实现，我们就可以在 Python shell 中轻易生成头像的 URL 了：

```
(venv) $ python manage.py shell
>>> u = User(email='john@example.com')
>>> u.gravatar()
'http://www.gravatar.com/avatar/d4c74594d84113932869575bd6?s=100&d=identicon&r=g'
>>> u.gravatar(size=256)
'http://www.gravatar.com/avatar/d4c74594d84113932869575bd6?s=256&d=identicon&r=g'
```

`gravatar()` 方法也可在 Jinja2 模板中调用。示例 10-14 在资料页面中添加了一个大小为 256 像素的头像。

示例 10-14 app/tempaltes/user.html：资料页面中的头像

```
...
<img class="img-rounded profile-thumbnail" src="{{ user.gravatar(size=256) }}">
...
```

使用类似方式，我们可在基模板的导航条上添加一个已登录用户头像的小型缩略图。为了更好地调整页面中头像图片的显示格式，我们可使用一些自定义的 CSS 类。你可以在源码仓库的 styles.css 文件中查看自定义的 CSS，styles.css 文件保存在程序静态文件的文件夹中，而且要在 base.html 模板中引用。图 10-3 为显示了头像的用户资料页面。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 10c` 签出程序的这个版本。

生成头像时要生成 MD5 值，这是一项 CPU 密集型操作。如果要在某个页面中生成大量头像，计算量会非常大。由于用户电子邮件地址的 MD5 散列值是不变的，因此可以将其缓存在 `User` 模型中。若要把 MD5 散列值保存在数据库中，需要对 `User` 模型做些改动，如示例 10-15 所示。

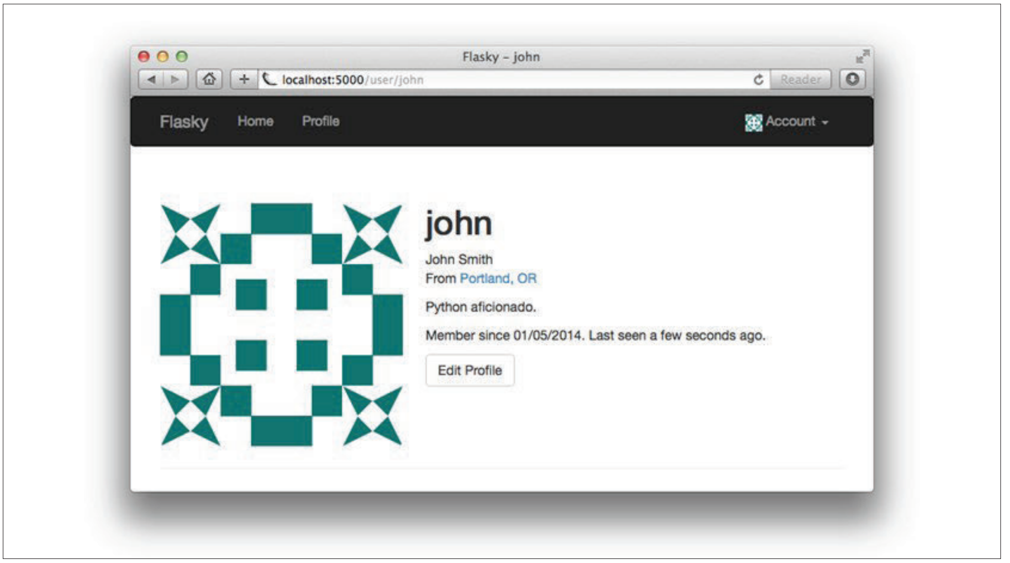


图 10-3　显示了头像的用户资料页面

示例 10-15　app/models.py：使用缓存的 MD5 散列值生成 Gravatar URL

```
class User(UserMixin, db.Model):
    # ...
    avatar_hash = db.Column(db.String(32))

    def __init__(self, **kwargs):
        # ...
        if self.email is not None and self.avatar_hash is None:
            self.avatar_hash = hashlib.md5(
                self.email.encode('utf-8')).hexdigest()

    def change_email(self, token):
        # ...
        self.email = new_email
        self.avatar_hash = hashlib.md5(
            self.email.encode('utf-8')).hexdigest()
        db.session.add(self)
        return True

    def gravatar(self, size=100, default='identicon', rating='g'):
        if request.is_secure:
            url = 'https://secure.gravatar.com/avatar'
        else:
            url = 'http://www.gravatar.com/avatar'
        hash = self.avatar_hash or hashlib.md5(
            self.email.encode('utf-8')).hexdigest()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

模型初始化过程中会计算电子邮件的散列值，然后存入数据库，若用户更新了电子邮件地址，则会重新计算散列值。`gravatar()` 方法会使用模型中保存的散列值；如果模型中没有，就和之前一样计算电子邮件地址的散列值。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 10d` 签出程序的这个版本。这个版本中包含了一个数据库迁移，签出代码后记得要运行 `python manage.py db upgrade`。

下一章，我们会创建这个程序使用的博客引擎。

第 11 章

博客文章

在本章，我们要实现 Flasky 的主要功能，即允许用户阅读、撰写博客文章。本章你会学到一些新技术：重用模板、分页显示长列表以及处理富文本。

11.1　提交和显示博客文章

为支持博客文章，我们需要创建一个新的数据库模型，如示例 11-1 所示。

示例 11-1　app/models.py：文章模型

```
class Post(db.Model):
    __tablename__ = 'posts'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    # ...
    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

博客文章包含正文、时间戳以及和 `User` 模型之间的一对多关系。`body` 字段的定义类型是 `db.Text`，所以不限制长度。

在程序的首页要显示一个表单，以便让用户写博客。这个表单很简单，只包括一个多行文本输入框，用于输入博客文章的内容，另外还有一个提交按钮，表单定义如示例 11-2 所示。

示例 11-2　app/main/forms.py：博客文章表单

```
class PostForm(Form):
```

```
        body = TextAreaField("What's on your mind?", validators=[Required()])
        submit = SubmitField('Submit')
```

index() 视图函数处理这个表单并把以前发布的博客文章列表传给模板，如示例 11-3 所示。

示例 11-3　app/main/views.py：处理博客文章的首页路由

```
@main.route('/', methods=['GET', 'POST'])
def index():
    form = PostForm()
    if current_user.can(Permission.WRITE_ARTICLES) and \
            form.validate_on_submit():
        post = Post(body=form.body.data,
                    author=current_user._get_current_object())
        db.session.add(post)
        return redirect(url_for('.index'))
    posts = Post.query.order_by(Post.timestamp.desc()).all()
    return render_template('index.html', form=form, posts=posts)
```

这个视图函数把表单和完整的博客文章列表传给模板。文章列表按照时间戳进行降序排列。博客文章表单采取惯常处理方式，如果提交的数据能通过验证就创建一个新 Post 实例。在发布新文章之前，要检查当前用户是否有写文章的权限。

注意，新文章对象的 author 属性值为表达式 current_user._get_current_object()。变量 current_user 由 Flask-Login 提供，和所有上下文变量一样，也是通过线程内的代理对象实现。这个对象的表现类似用户对象，但实际上却是一个轻度包装，包含真正的用户对象。数据库需要真正的用户对象，因此要调用 _get_current_object() 方法。

这个表单显示在 index.html 模板中欢迎消息的下方，其后是博客文章列表。在这个博客文章列表中，我们首次尝试创建博客文章时间轴，按照时间顺序由新到旧列出了数据库中所有的博客文章。对模板所做的改动如示例 11-4 所示。

示例 11-4　app/templates/index.html：显示博客文章的首页模板

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
...
<div>
    {% if current_user.can(Permission.WRITE_ARTICLES) %}
    {{ wtf.quick_form(form) }}
    {% endif %}
</div>
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        <div class="profile-thumbnail">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                <img class="img-rounded profile-thumbnail"
                    src="{{ post.author.gravatar(size=40) }}">
            </a>
        </div>
```

```
            <div class="post-date">{{ moment(post.timestamp).fromNow() }}</div>
            <div class="post-author">
                <a href="{{ url_for('.user', username=post.author.username) }}">
                    {{ post.author.username }}
                </a>
            </div>
            <div class="post-body">{{ post.body }}</div>
        </li>
        {% endfor %}
    </ul>
    ...
```

注意，如果用户所属角色没有 `WRITE_ARTICLES` 权限，则经 `User.can()` 方法检查后，不会显示博客文章表单。博客文章列表通过 HTML 无序列表实现，并指定了一个 CSS 类，从而让格式更精美。页面左侧会显示作者的小头像，头像和作者用户名都渲染成链接形式，可链接到用户资料页面。所用的 CSS 样式都存储在程序 static 文件夹里的 style.css 文件中。你可到 GitHub 仓库中查看这个文件。显示有表单和博客文章列表的首页如图 11-1 所示。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 11a` 签出程序的这个版本。这个版本包含了一个数据库迁移，签出代码后记得要运行 `python manage.py db upgrade`。

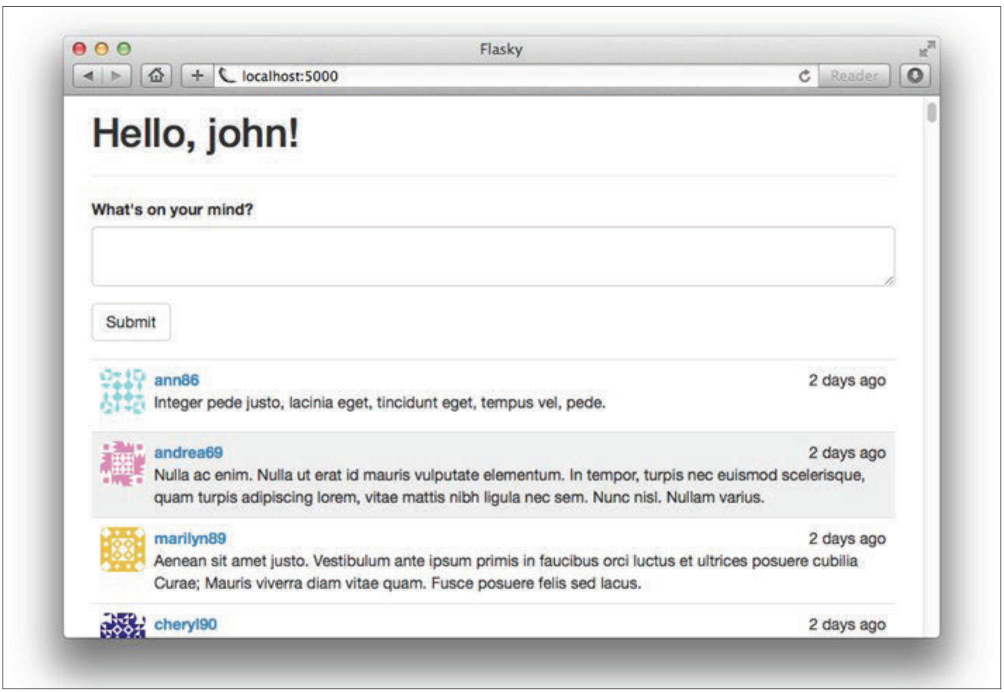


图 11-1　显示有博客发布表单和博客文章列表的首页

11.2　在资料页中显示博客文章

我们可以将用户资料页改进一下，在上面显示该用户发布的博客文章列表。示例 11-5 是对视图函数所做的改动，用以获取文章列表。

示例 11-5　app/main/views.py：获取博客文章的资料页路由

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        abort(404)
    posts = user.posts.order_by(Post.timestamp.desc()).all()
    return render_template('user.html', user=user, posts=posts)
```

用户发布的博客文章列表通过 `User.posts` 关系获取，`User.posts` 返回的是查询对象，因此可在其上调用过滤器，例如 `order_by()`。

和 index.html 模板一样，user.html 模板也要使用一个 HTML `<ul>` 元素渲染博客文章。维护两个完全相同的 HTML 片段副本可不是个好主意，遇到这种情况，Jinja2 提供的 `include()` 指令就非常有用。user.html 模板包含了其他文件中定义的列表，如示例 11-6 所示。

示例 11-6　app/templates/user.html：显示有博客文章的资料页模板

```
...
<h3>Posts by {{ user.username }}</h3>
{% include '_posts.html' %}
...
```

为了完成这种新的模板组织方式，index.html 模板中的 `<ul>` 元素需要移到新模板 _posts.html 中，并替换成另一个 `include()` 指令。注意，_posts.html 模板名的下划线前缀不是必须使用的，这只是一种习惯用法，以区分独立模板和局部模板。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 11b` 签出程序的这个版本。

11.3　分页显示长博客文章列表

随着网站的发展，博客文章的数量会不断增多，如果要在首页和资料页显示全部文章，浏览速度会变慢且不符合实际需求。在 Web 浏览器中，内容多的网页需要花费更多的时间生成、下载和渲染，所以网页内容变多会降低用户体验的质量。这一问题的解决方法是分页显示数据，进行片段式渲染。

11.3.1 创建虚拟博客文章数据

若想实现博客文章分页，我们需要一个包含大量数据的测试数据库。手动添加数据库记录浪费时间而且很麻烦，所以最好能使用自动化方案。有多个 Python 包可用于生成虚拟信息，其中功能相对完善的是 ForgeryPy，可以使用 `pip` 进行安装：

```
(venv) $ pip install forgerypy
```

严格来说，ForgeryPy 并不是这个程序的依赖，因为它只在开发过程中使用。为了区分生产环境的依赖和开发环境的依赖，我们可以把文件 requirements.txt 换成 requirements 文件夹，它们分别保存不同环境中的依赖。在这个新建的文件夹中，我们可以创建一个 dev.txt 文件，列出开发过程中所需的依赖，再创建一个 prod.txt 文件，列出生产环境所需的依赖。由于两个环境所需的依赖大部分是相同的，因此可以创建一个 common.txt 文件，在 dev.txt 和 prod.txt 中使用 `-r` 参数导入。dev.txt 文件的内容如示例 11-7 所示。

示例 11-7 requirements/dev.txt：开发所需的依赖文件

```
-r common.txt
ForgeryPy==0.1
```

示例 11-8 展示了添加到 `User` 模型和 `Post` 模型中的类方法，用来生成虚拟数据。

示例 11-8 app/models.py：生成虚拟用户和博客文章

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def generate_fake(count=100):
        from sqlalchemy.exc import IntegrityError
        from random import seed
        import forgery_py

        seed()
        for i in range(count):
            u = User(email=forgery_py.internet.email_address(),
                     username=forgery_py.internet.user_name(True),
                     password=forgery_py.lorem_ipsum.word(),
                     confirmed=True,
                     name=forgery_py.name.full_name(),
                     location=forgery_py.address.city(),
                     about_me=forgery_py.lorem_ipsum.sentence(),
                     member_since=forgery_py.date.date(True))
            db.session.add(u)
            try:
                db.session.commit()
            except IntegrityError:
                db.session.rollback()

class Post(db.Model):
    # ...
    @staticmethod
```

```
    def generate_fake(count=100):
        from random import seed, randint
        import forgery_py

        seed()
        user_count = User.query.count()
        for i in range(count):
            u = User.query.offset(randint(0, user_count - 1)).first()
            p = Post(body=forgery_py.lorem_ipsum.sentences(randint(1, 3)),
                     timestamp=forgery_py.date.date(True),
                     author=u)
            db.session.add(p)
            db.session.commit()
```

这些虚拟对象的属性由 ForgeryPy 的随机信息生成器生成，其中的名字、电子邮件地址、句子等属性看起来就像真的一样。

用户的电子邮件地址和用户名必须是唯一的，但 ForgeryPy 随机生成这些信息，因此有重复的风险。如果发生了这种不太可能出现的情况，提交数据库会话时会抛出 `IntegrityError` 异常。这个异常的处理方式是，在继续操作之前回滚会话。在循环中生成重复内容时不会把用户写入数据库，因此生成的虚拟用户总数可能会比预期少。

随机生成文章时要为每篇文章随机指定一个用户。为此，我们使用 `offset()` 查询过滤器。这个过滤器会跳过参数中指定的记录数量。通过设定一个随机的偏移值，再调用 `first()` 方法，就能每次都获得一个不同的随机用户。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 11c` 签出程序的这个版本。为保证安装了所有依赖，我们还要运行 `pip install -r requirements/dev.txt`。

使用新添加的方法，我们可以在 Python shell 中轻易生成大量虚拟用户和文章：

```
(venv) $ python manage.py shell
>>> User.generate_fake(100)
>>> Post.generate_fake(100)
```

如果你现在运行程序，会看到首页中显示了一个很长的随机博客文章列表。

11.3.2 在页面中渲染数据

示例 11-9 展示了为支持分页对首页路由所做的改动。

示例 11-9 app/main/views.py：分页显示博客文章列表

```
@main.route('/', methods=['GET', 'POST'])
```

```
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
                           pagination=pagination)
```

渲染的页数从请求的查询字符串（request.args）中获取，如果没有明确指定，则默认渲染第一页。参数 type=int 保证参数无法转换成整数时，返回默认值。

为了显示某页中的记录，要把 all() 换成 Flask-SQLAlchemy 提供的 paginate() 方法。页数是 paginate() 方法的第一个参数，也是唯一必需的参数。可选参数 per_page 用来指定每页显示的记录数量；如果没有指定，则默认显示 20 个记录。另一个可选参数为 error_out，当其设为 True 时（默认值），如果请求的页数超出了范围，则会返回 404 错误；如果设为 False，页数超出范围时会返回一个空列表。为了能够很便利地配置每页显示的记录数量，参数 per_page 的值从程序的环境变量 FLASKY_POSTS_PER_PAGE 中读取。

这样修改之后，首页中的文章列表只会显示有限数量的文章。若想查看第 2 页中的文章，要在浏览器地址栏中的 URL 后加上查询字符串 ?page=2。

11.3.3 添加分页导航

paginate() 方法的返回值是一个 Pagination 类对象，这个类在 Flask-SQLAlchemy 中定义。这个对象包含很多属性，用于在模板中生成分页链接，因此将其作为参数传入了模板。分页对象的属性简介如表 11-1 所示。

表11-1 Flask-SQLAlchemy分页对象的属性

属　　性	说　　明
items	当前页面中的记录
query	分页的源查询
page	当前页数
prev_num	上一页的页数
next_num	下一页的页数
has_next	如果有下一页，返回 True
has_prev	如果有上一页，返回 True
pages	查询得到的总页数
per_page	每页显示的记录数量
total	查询返回的记录总数

在分页对象上还可调用一些方法，如表 11-2 所示。

表11-2　在Flask-SQLAlchemy对象上可调用的方法

方　　法	说　　明
iter_pages (left_edge=2, left_current=2, right_current=5, right_edge=2)	一个迭代器，返回一个在分页导航中显示的页数列表。这个列表的最左边显示 left_edge 页，当前页的左边显示 left_current 页，当前页的右边显示 right_current 页，最右边显示 right_edge 页。例如，在一个 100 页的列表中，当前页为第 50 页，使用默认配置，这个方法会返回以下页数：1、2、None、48、49、50、51、52、53、54、55、None、99、100。None 表示页数之间的间隔
prev()	上一页的分页对象
next()	下一页的分页对象

拥有这么强大的对象和 Bootstrap 中的分页 CSS 类，我们很轻易地就能在模板底部构建一个分页导航。示例 11-10 是以 Jinja2 宏的形式实现的分页导航。

示例 11-10　app/templates/_macros.html：分页模板宏

```
{% macro pagination_widget(pagination, endpoint) %}
<ul class="pagination">
    <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_prev %}{{ url_for(endpoint,
            page = pagination.page - 1, **kwargs) }}{% else %}#{% endif %}">
            &laquo;
        </a>
    </li>
    {% for p in pagination.iter_pages() %}
        {% if p %}
            {% if p == pagination.page %}
            <li class="active">
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% else %}
            <li>
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% endif %}
        {% else %}
        <li class="disabled"><a href="#">&hellip;</a></li>
        {% endif %}
    {% endfor %}
    <li{% if not pagination.has_next %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_next %}{{ url_for(endpoint,
            page = pagination.page + 1, **kwargs) }}{% else %}#{% endif %}">
            &raquo;
        </a>
    </li>
</ul>
{% endmacro %}
```

这个宏创建了一个 Bootstrap 分页元素，即一个有特殊样式的无序列表，其中定义了下述页面链接。

- “上一页”链接。如果当前页是第一页，则为这个链接加上 disabled 类。

- 分页对象的 iter_pages() 迭代器返回的所有页面链接。这些页面被渲染成具有明确页数的链接，页数在 url_for() 的参数中指定。当前显示的页面使用 activeCSS 类高亮显示。页数列表中的间隔使用省略号表示。
- “下一页”链接。如果当前页是最后一页，则会禁用这个链接。

Jinja2 宏的参数列表中不用加入 **kwargs 即可接收关键字参数。分页宏把接收到的所有关键字参数都传给了生成分页链接的 url_for() 方法。这种方式也可在路由中使用，例如包含一个动态部分的资料页。

pagination_widget 宏可放在 index.html 和 user.html 中的 _posts.html 模板后面。示例 11-11 是它在程序首页中的应用。

示例 11-11　app/templates/index.html：在博客文章列表下面添加分页导航

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% import "_macros.html" as macros %}
...
{% include '_posts.html' %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.index') }}
</div>
{% endif %}
```

页面中的分页链接如图 11-2 所示。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 git checkout 11d 签出程序的这个版本。

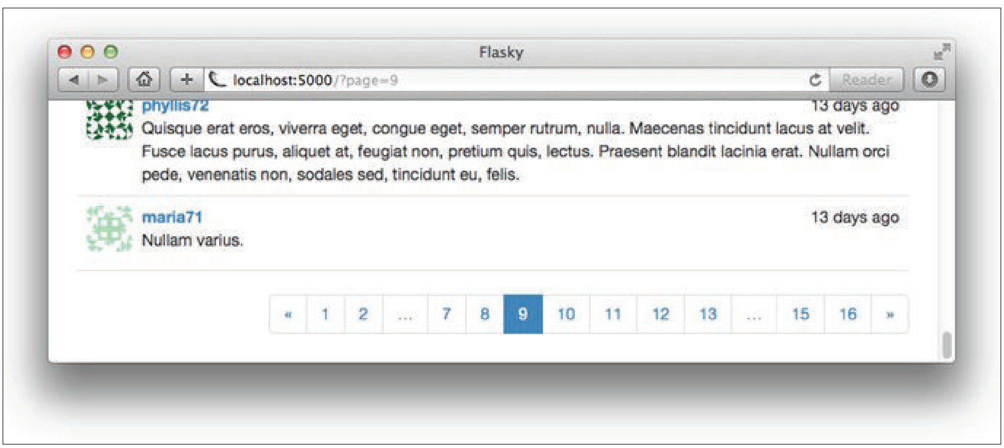


图 11-2　博客文章分页链接

11.4　使用Markdown和Flask-PageDown支持富文本文章

对于发布短消息和状态更新来说，纯文本足够用了，但如果用户想发布长文章，就会觉得在格式上受到了限制。本节我们要将输入文章的多行文本输入框升级，让其支持 Markdown（http://daringfireball.net/projects/markdown/）语法，还要添加富文本文章的预览功能。

实现这个功能要用到一些新包。

- PageDown：使用 JavaScript 实现的客户端 Markdown 到 HTML 的转换程序。
- Flask-PageDown：为 Flask 包装的 PageDown，把 PageDown 集成到 Flask-WTF 表单中。
- Markdown：使用 Python 实现的服务器端 Markdown 到 HTML 的转换程序。
- Bleach：使用 Python 实现的 HTML 清理器。

这些 Python 包可使用 `pip` 安装：

```
(venv) $ pip install flask-pagedown markdown bleach
```

11.4.1　使用Flask-PageDown

Flask-PageDown 扩展定义了一个 `PageDownField` 类，这个类和 WTForms 中的 `TextAreaField` 接口一致。使用 `PageDownField` 字段之前，先要初始化扩展，如示例 11-12 所示。

示例 11-12　app/__init__.py：初始化 Flask-PageDown

```
from flask.ext.pagedown import PageDown
# ...
pagedown = PageDown()
# ...
def create_app(config_name):
    # ...
    pagedown.init_app(app)
    # ...
```

若想把首页中的多行文本控件转换成 Markdown 富文本编辑器，`PostForm` 表单中的 `body` 字段要进行修改，如示例 11-13 所示。

示例 11-13　app/main/forms.py：启用 Markdown 的文章表单

```
from flask.ext.pagedown.fields import PageDownField

class PostForm(Form):
    body = PageDownField("What's on your mind?", validators=[Required()])
    submit = SubmitField('Submit')
```

Markdown 预览使用 PageDown 库生成，因此要在模板中修改。Flask-PageDown 简化了这

个过程，提供了一个模板宏，从 CDN 中加载所需文件，如示例 11-14 所示。

示例 11-14　app/index.html：Flask-PageDown 模板声明

```
{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 11e` 签出程序的这个版本。为保证安装了所有依赖，请执行 `pip install -r requirements/dev.txt`。

做了上述修改后，在多行文本字段中输入 Markdown 格式的文本会被立即渲染成 HTML 并显示在输入框下方。富文本博客文章表单如图 11-3 所示。

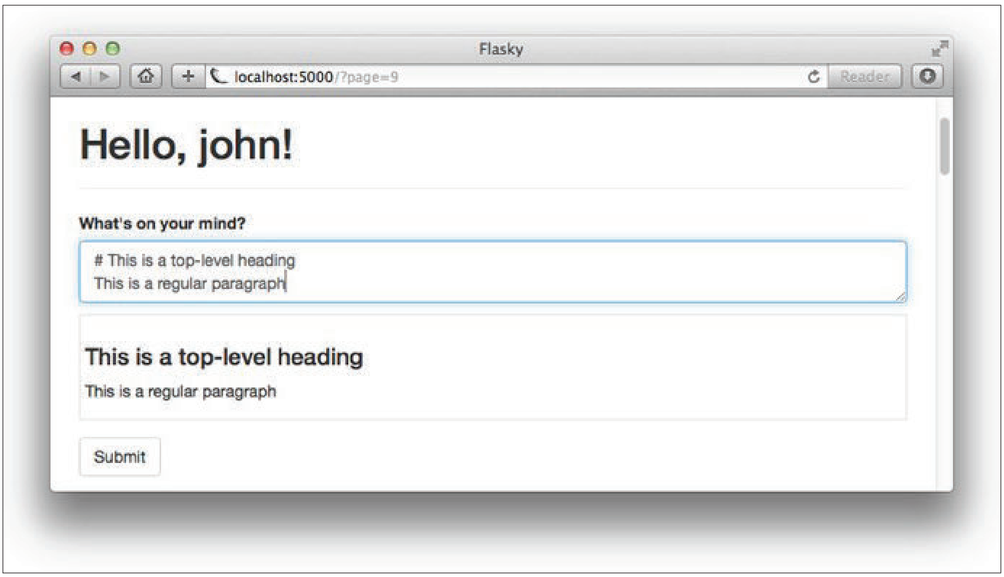


图 11-3　富文本博客文章表单

11.4.2　在服务器上处理富文本

提交表单后，`POST` 请求只会发送纯 Markdown 文本，页面中显示的 HTML 预览会被丢掉。和表单一起发送生成的 HTML 预览有安全隐患，因为攻击者轻易就能修改 HTML 代码，让其和 Markdown 源不匹配，然后再提交表单。安全起见，只提交 Markdown 源文本，在服务器上使用 Markdown（使用 Python 编写的 Markdown 到 HTML 转换程序）将其转换成 HTML。得到 HTML 后，再使用 Bleach 进行清理，确保其中只包含几个允许使用的

HTML 标签。

把 Markdown 格式的博客文章转换成 HTML 的过程可以在 _posts.html 模板中完成，但这么做效率不高，因为每次渲染页面时都要转换一次。为了避免重复工作，我们可在创建博客文章时做一次性转换。转换后的博客文章 HTML 代码缓存在 `Post` 模型的一个新字段中，在模板中可以直接调用。文章的 Markdown 源文本还要保存在数据库中，以防需要编辑。示例 11-15 是对 `Post` 模型所做的改动。

示例 11-15 app/models.py：在 Post 模型中处理 Markdown 文本

```
from markdown import markdown
import bleach

class Post(db.Model):
    # ...
    body_html = db.Column(db.Text)

    # ...
    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
                        'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul',
                        'h1', 'h2', 'h3', 'p']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Post.body, 'set', Post.on_changed_body)
```

`on_changed_body` 函数注册在 `body` 字段上，是 SQLAlchemy“set”事件的监听程序，这意味着只要这个类实例的 `body` 字段设了新值，函数就会自动被调用。`on_changed_body` 函数把 `body` 字段中的文本渲染成 HTML 格式，结果保存在 `body_html` 中，自动且高效地完成 Markdown 文本到 HTML 的转换。

真正的转换过程分三步完成。首先，`markdown()` 函数初步把 Markdown 文本转换成 HTML。然后，把得到的结果和允许使用的 HTML 标签列表传给 `clean()` 函数。`clean()` 函数删除所有不在白名单中的标签。转换的最后一步由 `linkify()` 函数完成，这个函数由 Bleach 提供，把纯文本中的 URL 转换成适当的 `<a>` 链接。最后一步是很有必要的，因为 Markdown 规范没有为自动生成链接提供官方支持。PageDown 以扩展的形式实现了这个功能，因此在服务器上要调用 `linkify()` 函数。

最后，如果 `post.body_html` 字段存在，还要把 `post.body` 换成 `post.body_html`，如示例 11-16 所示。

示例 11-16 app/templates/_posts.html：在模板中使用文章内容的 HTML 格式

```
...
<div class="post-body">
```

```
        {% if post.body_html %}
            {{ post.body_html | safe }}
        {% else %}
            {{ post.body }}
        {% endif %}
    </div>
    ...
```

渲染 HTML 格式内容时使用 | safe 后缀，其目的是告诉 Jinja2 不要转义 HTML 元素。出于安全考虑，默认情况下 Jinja2 会转义所有模板变量。Markdown 转换成的 HTML 在服务器上生成，因此可以放心渲染。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 git checkout 11f 签出程序的这个版本。这个版本包含了一个数据库迁移，签出代码后记得要运行 python manage.py db upgrade。为保证你安装了所有依赖，还要执行 pip install -r requirements/dev.txt。

11.5　博客文章的固定链接

用户有时希望能在社交网络中和朋友分享某篇博客文章的链接。为此，每篇文章都要有一个专页，使用唯一的 URL 引用。支持固定链接功能的路由和视图函数如示例 11-17 所示。

示例 11-17　app/main/views.py：文章的固定链接

```
@main.route('/post/<int:id>')
def post(id):
    post = Post.query.get_or_404(id)
    return render_template('post.html', posts=[post])
```

博客文章的 URL 使用插入数据库时分配的唯一 id 字段构建。

> 某些类型的程序使用可读性高的字符串而不是数字 ID 构建固定链接。除了数字 ID 之外，程序还为博客文章起了个独特的字符串别名。

注意，post.html 模板接收一个列表作为参数，这个列表就是要渲染的文章。这里必须要传入列表，因为只有这样，index.html 和 user.html 引用的 _posts.html 模板才能在这个页面中使用。

固定链接添加到通用模板 _posts.html 中，显示在文章下方，如示例 11-18 所示。

示例 11-18　app/templates/_posts.html：文章的固定链接

```
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        ...
        <div class="post-content">
            ...
            <div class="post-footer">
                <a href="{{ url_for('.post', id=post.id) }}">
                    <span class="label label-default">Permalink</span>
                </a>
            </div>
        </div>
    </li>
    {% endfor %}
</ul>
```

渲染固定链接页面的 post.html 模板如示例 11-19 所示，其中引入了上述模板。

示例 11-19　app/templates/post.html：固定链接模板

```
{% extends "base.html" %}

{% block title %}Flasky - Post{% endblock %}

{% block page_content %}
{% include '_posts.html' %}
{% endblock %}
```

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 11g` 签出程序的这个版本。

11.6　博客文章编辑器

与博客文章相关的最后一个功能是文字编辑器，它允许用户编辑自己的文章。博客文章编辑器显示在单独的页面中。在这个页面的上部会显示文章的当前版本，以供参考，下面跟着一个 Markdown 编辑器，用于修改 Markdown 源。这个编辑器基于 Flask-PageDown 实现，所以页面下部还会显示一个编辑后的文章预览。edit_post.html 模板如示例 11-20 所示。

示例 11-20　app/templates/edit_post.html：编辑博客文章的模板

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Edit Post{% endblock %}
```

```
{% block page_content %}
<div class="page-header">
    <h1>Edit Post</h1>
</div>
<div>
    {{ wtf.quick_form(form) }}
</div>
{% endblock %}

{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

博客文章编辑器使用的路由如示例 11-21 所示。

示例 11-21 app/main/views.py：编辑博客文章的路由

```
@main.route('/edit/<int:id>', methods=['GET', 'POST'])
@login_required
def edit(id):
    post = Post.query.get_or_404(id)
    if current_user != post.author and \
            not current_user.can(Permission.ADMINISTER):
        abort(403)
    form = PostForm()
    if form.validate_on_submit():
        post.body = form.body.data
        db.session.add(post)
        flash('The post has been updated.')
        return redirect(url_for('post', id=post.id))
    form.body.data = post.body
    return render_template('edit_post.html', form=form)
```

这个视图函数的作用是只允许博客文章的作者编辑文章，但管理员例外，管理员能编辑所有用户的文章。如果用户试图编辑其他用户的文章，视图函数会返回 403 错误。这里使用的 `PostForm` 表单类和首页中使用的是同一个。

为了功能完整，我们还可以在每篇博客文章的下面、固定链接的旁边添加一个指向编辑页面的链接，如示例 11-22 所示。

示例 11-22 app/templates/_posts.html：编辑博客文章的链接

```
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        ...
        <div class="post-content">
            ...
            <div class="post-footer">
                ...
                {% if current_user == post.author %}
                <a href="{{ url_for('.edit', id=post.id) }}">
```

```
                <span class="label label-primary">Edit</span>
            </a>
            {% elif current_user.is_administrator() %}
            <a href="{{ url_for('.edit', id=post.id) }}">
                <span class="label label-danger">Edit [Admin]</span>
            </a>
            {% endif %}
        </div>
    </div>
</li>
{% endfor %}
</ul>
```

通过这次修改，我们在当前用户发布的博客文章下面添加了一个“Edit”链接。如果当前用户是管理员，所有文章下面都会有编辑链接。为管理员显示的链接样式有点不同，以从视觉上表明这是管理功能。图 11-4 是在浏览器中显示的编辑链接和固定链接。

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 11h` 签出程序的这个版本。

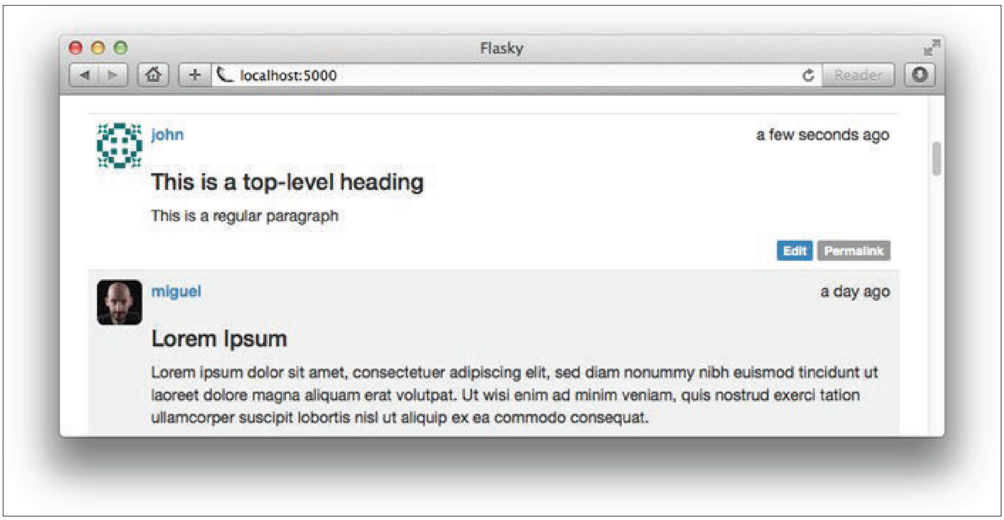


图 11-4　博客文章的编辑链接和固定链接

第 12 章

关注者

社交 Web 程序允许用户之间相互联系。在程序中，这种关系称为*关注者*、*好友*、*联系人*、*联络人*或*伙伴*。但不管使用哪个名字，其功能都是一样的，而且都要记录两个用户之间的定向联系，在数据库查询中也要使用这种联系。

在本章，你将学到如何在 Flasky 中实现关注功能，让用户“关注”其他用户，并在首页只显示所关注用户发布的博客文章列表。

12.1　再论数据库关系

我们在第 5 章介绍过，数据库使用*关系*建立记录之间的联系。其中，一对多关系是最常用的关系类型，它把一个记录和一组相关的记录联系在一起。实现这种关系时，要在“多”这一侧加入一个外键，指向“一”这一侧联接的记录。本书开发的示例程序现在包含两个一对多关系：一个把用户角色和一组用户联系起来，另一个把用户和发布的博客文章联系起来。

大部分的其他关系类型都可以从一对多类型中衍生。*多对一*关系从“多”这一侧看，就是一对多关系。一对一关系类型是简化版的一对多关系，限制“多”这一侧最多只能有一个记录。唯一不能从一对多关系中简单演化出来的类型是*多对多*关系，这种关系的两侧都有多个记录。下一节将详细介绍多对多关系。

12.1.1　多对多关系

一对多关系、多对一关系和一对一关系至少都有一侧是单个实体，所以记录之间的联系通过外键实现，让外键指向这个实体。但是，你要如何实现两侧都是“多”的关系呢？

下面以一个典型的多对多关系为例，即一个记录学生和他们所选课程的数据库。很显然，你不能在学生表中加入一个指向课程的外键，因为一个学生可以选择多个课程，一个外键不够用。同样，你也不能在课程表中加入一个指向学生的外键，因为一个课程有多个学生选择。两侧都需要一组外键。

这种问题的解决方法是添加第三张表，这个表称为*关联表*。现在，多对多关系可以分解成原表和关联表之间的两个一对多关系。图 12-1 描绘了学生和课程之间的多对多关系。

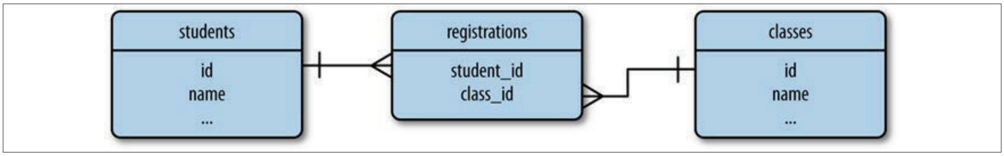


图 12-1　多对多关系示例

这个例子中的关联表是 `registrations`，表中的每一行都表示一个学生注册的一个课程。

查询多对多关系要分成两步。若想知道某位学生选择了哪些课程，你要先从学生和注册之间的一对多关系开始，获取这位学生在 `registrations` 表中的所有记录，然后再按照多到一的方向遍历课程和注册之间的一对多关系，找到这位学生在 `registrations` 表中各记录所对应的课程。同样，若想找到选择了某门课程的所有学生，你要先从课程表中开始，获取其在 `registrations` 表中的记录，再获取这些记录联接的学生。

通过遍历两个关系来获取查询结果的做法听起来有难度，不过像前例这种简单关系，SQLAlchemy 就可以完成大部分操作。图 12-1 中的多对多关系使用的代码表示如下：

```
registrations = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id'))
)

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    classes = db.relationship('Class',
                              secondary=registrations,
                              backref=db.backref('students', lazy='dynamic'),
                              lazy='dynamic')

class Class(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    name = db.Column(db.String)
```

多对多关系仍使用定义一对多关系的 `db.relationship()` 方法进行定义，但在多对多关系中，必须把 `secondary` 参数设为关联表。多对多关系可以在任何一个类中定义，`backref` 参数会处理好关系的另一侧。关联表就是一个简单的表，不是模型，SQLAlchemy 会自动接管这个表。

classes 关系使用列表语义，这样处理多对多关系特别简单。假设学生是 s，课程是 c，学生注册课程的代码为：

```
>>> s.classes.append(c)
>>> db.session.add(s)
```

列出学生 s 注册的课程以及注册了课程 c 的学生也很简单：

```
>>> s.classes.all()
>>> c.students.all()
```

Class 模型中的 students 关系由参数 db.backref() 定义。注意，这个关系中还指定了 lazy = 'dynamic' 参数，所以关系两侧返回的查询都可接受额外的过滤器。

如果后来学生 s 决定不选课程 c 了，那么可使用下面的代码更新数据库：

```
>>> s.classes.remove(c)
```

12.1.2 自引用关系

多对多关系可用于实现用户之间的关注，但存在一个问题。在学生和课程的例子中，关联表联接的是两个明确的实体。但是，表示用户关注其他用户时，只有用户一个实体，没有第二个实体。

如果关系中的两侧都在同一个表中，这种关系称为*自引用关系*。在关注中，关系的左侧是用户实体，可以称为“关注者”；关系的右侧也是用户实体，但这些是“被关注者”。从概念上来看，自引用关系和普通关系没什么区别，只是不易理解。图 12-2 是自引用关系的数据库图解，表示用户之间的关注。

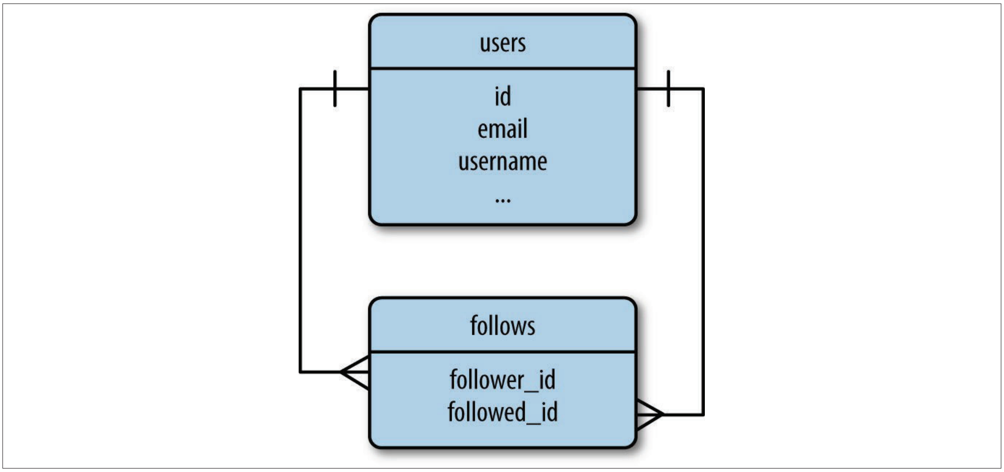


图 12-2　关注者，多对多关系

本例的关联表是 follows，其中每一行都表示一个用户关注了另一个用户。图中左边表示的一对多关系把用户和 follows 表中的一组记录联系起来，用户是关注者。图中右边表示的一对多关系把用户和 follows 表中的一组记录联系起来，用户是被关注者。

12.1.3　高级多对多关系

使用前一节介绍的自引用多对多关系可在数据库中表示用户之间的关注，但却有个限制。使用多对多关系时，往往需要存储所联两个实体之间的额外信息。对用户之间的关注来说，可以存储用户关注另一个用户的日期，这样就能按照时间顺序列出所有关注者。这种信息只能存储在关联表中，但是在之前实现的学生和课程之间的关系中，关联表完全是由 SQLAlchemy 掌控的内部表。

为了能在关系中处理自定义的数据，我们必须提升关联表的地位，使其变成程序可访问的模型。新的关联表如示例 12-1 所示，使用 Follow 模型表示。

示例 12-1　app/models/user.py：关注关联表的模型实现

```
class Follow(db.Model):
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    followed_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

SQLAlchemy 不能直接使用这个关联表，因为如果这么做程序就无法访问其中的自定义字段。相反地，要把这个多对多关系的左右两侧拆分成两个基本的一对多关系，而且要定义成标准的关系。代码如示例 12-2 所示。

示例 12-2　app/models/user.py：使用两个一对多关系实现的多对多关系

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship('Follow',
                               foreign_keys=[Follow.follower_id],
                               backref=db.backref('follower', lazy='joined'),
                               lazy='dynamic',
                               cascade='all, delete-orphan')
    followers = db.relationship('Follow',
                                foreign_keys=[Follow.followed_id],
                                backref=db.backref('followed', lazy='joined'),
                                lazy='dynamic',
                                cascade='all, delete-orphan')
```

在这段代码中，followed 和 followers 关系都定义为单独的一对多关系。注意，为了消除外键间的歧义，定义关系时必须使用可选参数 foreign_keys 指定的外键。而且，db.backref() 参数并不是指定这两个关系之间的引用关系，而是回引 Follow 模型。

回引中的 lazy 参数指定为 joined。这个 lazy 模式可以实现立即从联结查询中加载相关对象。例如，如果某个用户关注了 100 个用户，调用 user.followed.all() 后会返回一个列表，其中包含 100 个 Follow 实例，每一个实例的 follower 和 followed 回引属性都指向相应的用户。设定为 lazy='joined' 模式，就可在一次数据库查询中完成这些操作。如果把 lazy 设为默认值 select，那么首次访问 follower 和 followed 属性时才会加载对应的用户，而且每个属性都需要一个单独的查询，这就意味着获取全部被关注用户时需要增加 100 次额外的数据库查询。

这两个关系中，User 一侧设定的 lazy 参数作用不一样。lazy 参数都在“一”这一侧设定，返回的结果是“多”这一侧中的记录。上述代码使用的是 dynamic，因此关系属性不会直接返回记录，而是返回查询对象，所以在执行查询之前还可以添加额外的过滤器。

cascade 参数配置在父对象上执行的操作对相关对象的影响。比如，层叠选项可设定为：将用户添加到数据库会话后，要自动把所有关系的对象都添加到会话中。层叠选项的默认值能满足大多数情况的需求，但对这个多对多关系来说却不合用。删除对象时，默认的层叠行为是把对象联接的所有相关对象的外键设为空值。但在关联表中，删除记录后正确的行为应该是把指向该记录的实体也删除，因为这样能有效销毁联接。这就是层叠选项值 delete-orphan 的作用。

> cascade 参数的值是一组由逗号分隔的层叠选项，这看起来可能让人有点困惑，但 all 表示除了 delete-orphan 之外的所有层叠选项。设为 all, delete-orphan 的意思是启用所有默认层叠选项，而且还要删除孤儿记录。

程序现在要处理两个一对多关系，以便实现多对多关系。由于这些操作经常需要重复执行，所以最好在 User 模型中为所有可能的操作定义辅助方法。用于控制关系的 4 个新方法如示例 12-3 所示。

示例 12-3 app/models.py：关注关系的辅助方法

```
class User(db.Model):
    # ...
    def follow(self, user):
        if not self.is_following(user):
            f = Follow(follower=self, followed=user)
            db.session.add(f)

    def unfollow(self, user):
        f = self.followed.filter_by(followed_id=user.id).first()
        if f:
            db.session.delete(f)

    def is_following(self, user):
        return self.followed.filter_by(
```

```
            followed_id=user.id).first() is not None

    def is_followed_by(self, user):
        return self.followers.filter_by(
            follower_id=user.id).first() is not None
```

follow() 方法手动把 Follow 实例插入关联表，从而把关注者和被关注者联接起来，并让程序有机会设定自定义字段的值。联接在一起的两个用户被手动传入 Follow 类的构造器，创建一个 Follow 新实例，然后像往常一样，把这个实例对象添加到数据库会话中。注意，这里无需手动设定 timestamp 字段，因为定义字段时指定了默认值，即当前日期和时间。unfollow() 方法使用 followed 关系找到联接用户和被关注用户的 Follow 实例。若要销毁这两个用户之间的联接，只需删除这个 Follow 对象即可。is_following() 方法和 is_followed_by() 方法分别在左右两边的一对多关系中搜索指定用户，如果找到了就返回 True。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，难么可以执行 git checkout 12a 签出程序的这个版本。这个版本包含了一个数据库迁移，签出代码后记得要运行 python manage.py db upgrade。

现在，关注功能在数据库中的部分完成了。你可以在 GitHub 上的源码仓库找到对于这个数据库关系的单元测试。

12.2　在资料页中显示关注者

如果用户查看一个尚未关注用户的资料页，页面中要显示一个“Follow”（关注）按钮，如果查看已关注用户的资料页则显示“Unfollow”（取消关注）按钮。而且，页面中最好能显示出关注者和被关注者的数量，再列出关注和被关注的用户列表，并在相应的用户资料页中显示“Follows You”（关注了你）标志。对用户资料页模板的改动如示例 12-4 所示。添加这些信息后的资料页如图 12-3 所示。

示例 12-4　app/templates/user.html：在用户资料页上部添加关注信息

```
{% if current_user.can(Permission.FOLLOW) and user != current_user %}
    {% if not current_user.is_following(user) %}
    <a href="{{ url_for('.follow', username=user.username) }}"
        class="btn btn-primary">Follow</a>
    {% else %}
    <a href="{{ url_for('.unfollow', username=user.username) }}"
        class="btn btn-default">Unfollow</a>
    {% endif %}
{% endif %}
<a href="{{ url_for('.followers', username=user.username) }}">
    Followers: <span class="badge">{{ user.followers.count() }}</span>
</a>
<a href="{{ url_for('.followed_by', username=user.username) }}">
```

```
    Following: <span class="badge">{{ user.followed.count() }}</span>
</a>
{% if current_user.is_authenticated() and user != current_user and
    user.is_following(current_user) %}
| <span class="label label-default">Follows you</span>
{% endif %}
```

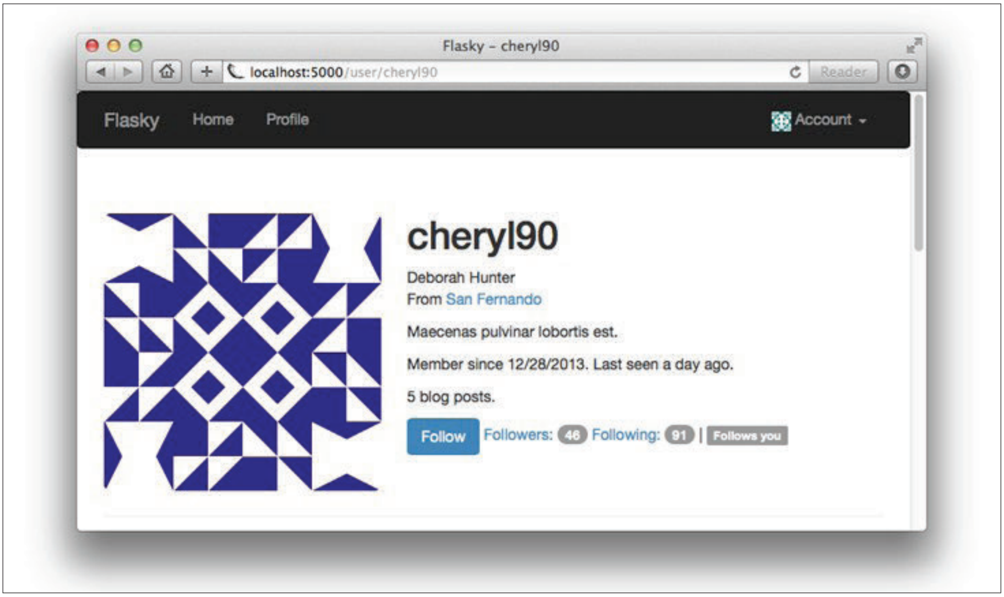


图 12-3　资料页中显示的关注信息

这次修改模板用到了 4 个新端点。用户在其他用户的资料页中点击“Follow”（关注）按钮后，执行的是 /follow/〈*username*〉路由。这个路由的实现方法如示例 12-5。

示例 12-5　app/main/views.py：“关注”路由和视图函数

```
@main.route('/follow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    if current_user.is_following(user):
        flash('You are already following this user.')
        return redirect(url_for('.user', username=username))
    current_user.follow(user)
    flash('You are now following %s.' % username)
    return redirect(url_for('.user', username=username))
```

这个视图函数先加载请求的用户，确保用户存在且当前登录用户还没有关注这个用户，然

后调用 `User` 模型中定义的辅助方法 `follow()`，用以联接两个用户。/unfollow/<*username*> 路由的实现方式类似。

用户在其他用户的资料页中点击关注者数量后，将调用 /followers/<*username*> 路由。这个路由的实现如示例 12-6 所示。

示例 12-6 app/main/views.py："关注者"路由和视图函数

```
@main.route('/followers/<username>')
def followers(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    page = request.args.get('page', 1, type=int)
    pagination = user.followers.paginate(
        page, per_page=current_app.config['FLASKY_FOLLOWERS_PER_PAGE'],
        error_out=False)
    follows = [{'user': item.follower, 'timestamp': item.timestamp}
               for item in pagination.items]
    return render_template('followers.html', user=user, title="Followers of",
                           endpoint='.followers', pagination=pagination,
                           follows=follows)
```

这个函数加载并验证请求的用户，然后使用第 11 章中介绍的技术分页显示该用户的 `followers` 关系。由于查询关注者返回的是 `Follow` 实例列表，为了渲染方便，我们将其转换成一个新列表，列表中的各元素都包含 `user` 和 `timestamp` 字段。

渲染关注者列表的模板可以写的通用一些，以便能用来渲染关注的用户列表和被关注的用户列表。模板接收的参数包括用户对象、分页链接使用的端点、分页对象和查询结果列表。

`followed_by` 端点的实现过程几乎一样，唯一区别在于：用户列表从 `user.followed` 关系中获取。传入模板的参数也要进行相应调整。

followers.html 模板使用两列表格实现，左边一列用于显示用户名和头像，右边一列用于显示 Flask-Moment 时间戳。你可以在 GitHub 上的源码仓库中查看具体的实现代码。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 12b` 签出程序的这个版本。

12.3 使用数据库联结查询所关注用户的文章

程序首页目前按时间降序显示数据库中的所有文章。现在我们已经完成了关注功能，如果

能让用户选择只查看所关注用户发布的博客文章就更好了。

若想显示所关注用户发布的所有文章，第一步显然先要获取这些用户，然后获取各用户的文章，再按一定顺序排列，写入单独列表。可是这种方式的伸缩性不好，随着数据库不断变大，生成这个列表的工作量也不断增长，而且分页等操作也无法高效率完成。获取博客文章的高效方式是只用一次查询。

完成这个操作的数据库操作称为*联结*。联结操作用到两个或更多的数据表，在其中查找满足指定条件的记录组合，再把记录组合插入一个临时表中，这个临时表就是联结查询的结果。理解联结查询的最好方法是实例讲解。

表 12-1 是一个 `users` 表示例，表中有 3 个用户。

表12-1　users表

id	username
1	john
2	susan
3	david

表 12-2 是对应的 `posts` 表，表中有几篇博客文章。

表12-2　posts表

id	author_id	body
1	2	susan 的博客文章
2	1	john 的博客文章
3	3	david 的博客文章
4	1	john 的第 2 篇博客文章

最后，表 12-3 显示谁关注了谁。从这个表中你可以看出，john 关注了 david，susan 关注了 john，但 david 谁也没关注。

表12-3　follows表

follower_id	followed_id
1	3
2	1
2	3

若想获得 susuan 所关注用户发布的文章，就要合并 `posts` 表和 `follows` 表。首先过滤 `follows` 表，只留下关注者为 susuan 的记录，即上面表中的最后两行。然后过滤 `posts` 表，留下 `author_id` 和过滤后的 `follows` 表中 `followed_id` 相等的记录，把两次过滤结果合并，组成临时联结表，这样就能高效查询 susuan 所关注用户的文章列表。表 12-4 是联结操作

得到的结果。表中用来执行联结操作的列被加上了 * 标记。

表12-4　联结表

id	author_id*	body	follower_id	followed_id*
2	1	john 的博客文章	2	1
3	3	david 的博客文章	2	3
4	1	john 的第 2 篇博客文章	2	1

这个表中包含的博客文章都是用户 susan 所关注用户发布的。使用 Flask-SQLAlchemy 执行这个联结操作的查询相当复杂：

```
return db.session.query(Post).select_from(Follow).\
    filter_by(follower_id=self.id).\
    join(Post, Follow.followed_id == Post.author_id)
```

你在此之前见到的查询都是从所查询模型的 `query` 属性开始的。这种查询不能在这里使用，因为查询要返回 `posts` 记录，所以首先要做的操作是在 `follows` 表上执行过滤器。因此，这里使用了一种更基础的查询方式。为了完全理解上述查询，下面分别说明各部分：

- `db.session.query(Post)` 指明这个查询要返回 `Post` 对象；
- `select_from(Follow)` 的意思是这个查询从 `Follow` 模型开始；
- `filter_by(follower_id=self.id)` 使用关注用户过滤 `follows` 表；
- `join(Post, Follow.followed_id == Post.author_id)` 联结 `filter_by()` 得到的结果和 `Post` 对象。

调换过滤器和联结的顺序可以简化这个查询：

```
return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
    .filter(Follow.follower_id == self.id)
```

如果首先执行联结操作，那么这个查询就可以从 `Post.query` 开始，此时唯一需要使用的两个过滤器是 `join()` 和 `filter()`。但这两种查询是一样的吗？先执行联结操作再过滤看起来工作量会更大一些，但实际上这两种查询是等效的。SQLAlchemy 首先收集所有的过滤器，然后再以最高效的方式生成查询。这两种查询生成的原生 SQL 指令是一样的。我们要把后一种查询写入 `Post` 模型，如示例 12-7 所示。

示例 12-7　app/models.py：获取所关注用户的文章

```
class User(db.Model):
    # ...
    @property
    def followed_posts(self):
        return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
            .filter(Follow.follower_id == self.id)
```

注意，`followed_posts()` 方法定义为属性，因此调用时无需加 `()`。如此一来，所有关系的句法都一样了。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 12c` 签出程序的这个版本。

联结非常难理解，你可能需要在 shell 中多研究一下示例代码才能完全领悟。

12.4　在首页显示所关注用户的文章

现在，用户可以选择在首页显示所有用户的博客文章还是只显示所关注用户的文章了。示例 12-8 显示了如何实现这种选择。

示例 12-8　app/main/views.py：显示所有博客文章或只显示所关注用户的文章

```
@app.route('/', methods = ['GET', 'POST'])
def index():
    # ...
    show_followed = False
    if current_user.is_authenticated():
        show_followed = bool(request.cookies.get('show_followed', ''))
    if show_followed:
        query = current_user.followed_posts
    else:
        query = Post.query
    pagination = query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
                           show_followed=show_followed, pagination=pagination)
```

决定显示所有博客文章还是只显示所关注用户文章的选项存储在 cookie 的 `show_followed` 字段中，如果其值为非空字符串，则表示只显示所关注用户的文章。cookie 以 `request.cookies` 字典的形式存储在请求对象中。这个 cookie 的值会转换成布尔值，根据得到的值设定本地变量 `query` 的值。`query` 的值决定最终获取所有博客文章的查询，或是获取过滤后的博客文章查询。显示所有用户的文章时，要使用顶级查询 `Post.query`；如果限制只显示所关注用户的文章，要使用最近添加的 `User.followed_posts` 属性。然后将本地变量 `query` 中保存的查询进行分页，像往常一样将其传入模板。

`show_followed`cookie 在两个新路由中设定，如示例 12-9 所示。

示例 12-9　app/main/views.py：查询所有文章还是所关注用户的文章

```
@main.route('/all')
```

```
@login_required
def show_all():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '', max_age=30*24*60*60)
    return resp

@main.route('/followed')
@login_required
def show_followed():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24*60*60)
    return resp
```

指向这两个路由的链接添加在首页模板中。点击这两个链接后会为 show_followedcookie 设定适当的值，然后重定向到首页。

cookie 只能在响应对象中设置，因此这两个路由不能依赖 Flask，要使用 make_response() 方法创建响应对象。

set_cookie() 函数的前两个参数分别是 cookie 名和值。可选的 max_age 参数设置 cookie 的过期时间，单位为秒。如果不指定参数 max_age，浏览器关闭后 cookie 就会过期。在本例中，过期时间为 30 天，所以即便用户几天不访问程序，浏览器也会记住设定的值。

接下来我们要对模板做些改动，在页面上部添加两个导航选项卡，分别调用 /all 和 /followed 路由，并在会话中设定正确的值。你可在 GitHub 上的源码仓库中查看模板改动详情。改动后的首页如图 12-4 所示。

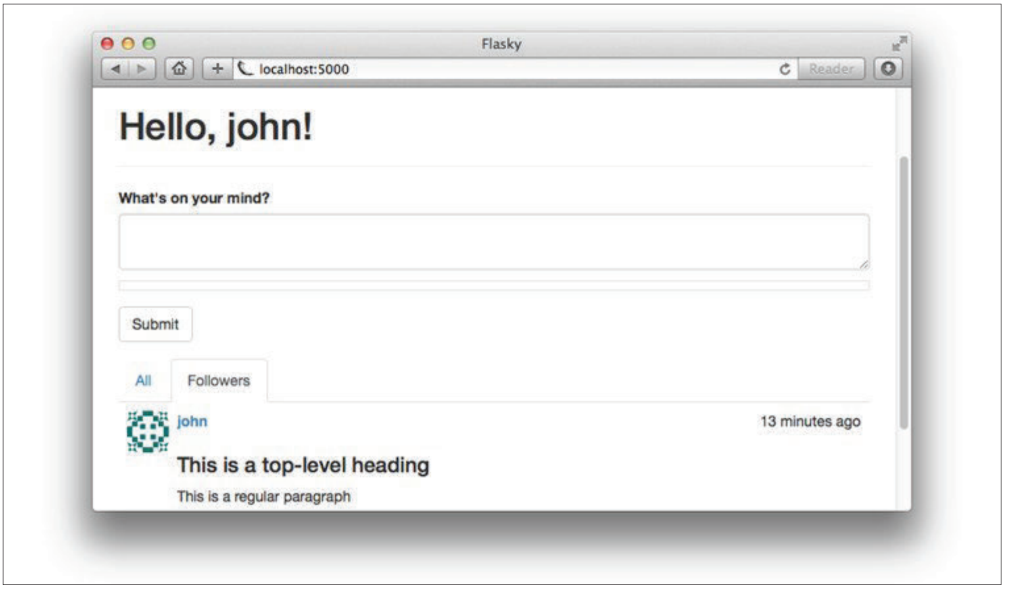


图 12-4　首页上显示的所关注用户文章

如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 12d` 签出程序的这个版本。

如果你现在访问网站，切换到所关注用户文章列表，会发现自己的文章不在列表中。这是肯定的，因为用户不能关注自己。

虽然查询能按设计正常执行，但用户查看好友文章时还是希望能看到自己的文章。这个问题最简单的解决办法是，注册时把用户设为自己的关注者。实现方法如示例 12-10 所示。

示例 12-10　app/models.py：构建用户时把用户设为自己的关注者

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

可是，现在的数据库中可能已经创建了一些用户，而且都没有关注自己。如果数据库还比较小，容易重新生成，那么可以删掉再重新创建。如果情况相反，那么正确的方法是添加一个函数，更新现有用户，如示例 12-11 所示。

示例 12-11　app/models.py：把用户设为自己的关注者

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def add_self_follows():
        for user in User.query.all():
            if not user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
    # ...
```

现在，可以通过在 shell 中运行这个函数来更新数据库：

```
(venv) $ python manage.py shell
>>> User.add_self_follows()
```

创建函数更新数据库这一技术经常用来更新已部署的程序，因为运行脚本更新比手动更新数据库更少出错。在第 17 章中，你会看到如何在部署脚本中使用这个函数及类似函数。

用户关注自己这一功能的实现让程序变得更实用，但也有一些副作用。因为用户的自关注链接，用户资料页显示的关注者和被关注者的数量都增加了 1 个。为了显示准确，这些数字要减去 1，这一点在模板中很容易实现，直接渲染 `{{ user.followers.count() - 1 }}` 和 `{{ user.followed.count() - 1 }}` 即可。然后，还要调整关注用户和被关注用户的列表，

不显示自己。这在模板中也容易实现，使用条件语句即可。最后，检查关注者数量的单元测试也会受到自关注的影响，必须做出调整，计入自关注。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 12e` 签出程序的这个版本。

下一章我们要实现用户评论子系统，这是社交程序的另一个重要功能。

第 13 章

用户评论

允许用户交互是社交博客平台成功的关键。在本章，你将学到如何实现用户评论。这里介绍的技术基本上可以直接用在大多数社交程序中。

13.1 评论在数据库中的表示

评论和博客文章没有太大区别，都有正文、作者和时间戳，而且在这个特定实现中都使用 Markdown 语法编写。图 13-1 是 `comments` 表的图解以及和其他数据表之间的关系。

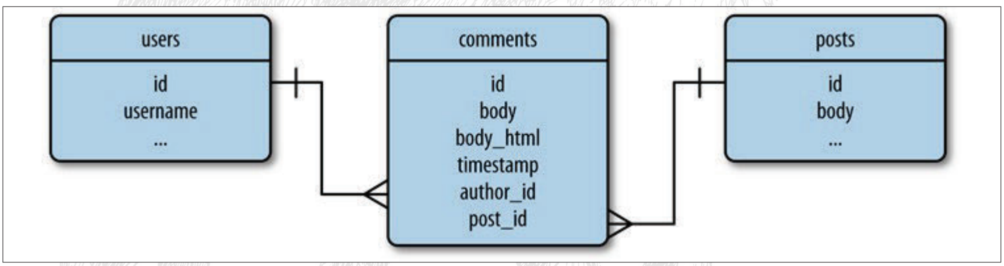


图 13-1 博客文章评论的数据库表示

评论属于某篇博客文章，因此定义了一个从 `posts` 表到 `comments` 表的一对多关系。使用这个关系可以获取某篇特定博客文章的评论列表。

`comments` 表还和 `users` 表之间有一对多关系。通过这个关系可以获取用户发表的所有评论，还能间接知道用户发表了多少篇评论。用户发表的评论数量可以显示在用户资料页中。`Comment` 模型的定义如示例 13-1。

示例 13-1　app/models.py：Comment 模型

```
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    disabled = db.Column(db.Boolean)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    post_id = db.Column(db.Integer, db.ForeignKey('posts.id'))

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i',
                        'strong']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

`Comment` 模型的属性几乎和 `Post` 模型一样，不过多了一个 `disabled` 字段。这是个布尔值字段，协管员通过这个字段查禁不当评论。和博客文章一样，评论也定义了一个事件，在修改 `body` 字段内容时触发，自动把 Markdown 文本转换成 HTML。转换过程和第 11 章中的博客文章一样，不过评论相对较短，而且对 Markdown 中允许使用的 HTML 标签要求更严格，要删除与段落相关的标签，只留下格式化字符的标签。

为了完成对数据库的修改，`User` 和 `Post` 模型还要建立与 `comments` 表的一对多关系，如示例 13-2 所示。

示例 13-2　app/models/user.py：users 和 posts 表与 comments 表之间的一对多关系

```
class User(db.Model):
    # ...
    comments = db.relationship('Comment', backref='author', lazy='dynamic')

class Post(db.Model):
    # ...
    comments = db.relationship('Comment', backref='post', lazy='dynamic')
```

13.2　提交和显示评论

在这个程序中，评论要显示在单篇博客文章页面中。这个页面在第 11 章添加固定链接时已经创建。在这个页面中还要有一个提交评论的表单。用来输入评论的表单如示例 13-3 所示。这个表单很简单，只有一个文本字段和一个提交按钮。

示例 13-3　app/main/forms.py：评论输入表单

```
class CommentForm(Form):
```

```
        body = StringField('', validators=[Required()])
        submit = SubmitField('Submit')
```

示例 13-4 是为了支持评论而更新的 /post/<*int:id*> 路由。

示例 13-4 app/main/views.py：支持博客文章评论

```
    @main.route('/post/<int:id>', methods=['GET', 'POST'])
    def post(id):
        post = Post.query.get_or_404(id)
        form = CommentForm()
        if form.validate_on_submit():
            comment = Comment(body=form.body.data,
                              post=post,
                              author=current_user._get_current_object())
            db.session.add(comment)
            flash('Your comment has been published.')
            return redirect(url_for('.post', id=post.id, page=-1))
        page = request.args.get('page', 1, type=int)
        if page == -1:
            page = (post.comments.count() - 1) / \
                   current_app.config['FLASKY_COMMENTS_PER_PAGE'] + 1
        pagination = post.comments.order_by(Comment.timestamp.asc()).paginate(
            page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
            error_out=False)
        comments = pagination.items
        return render_template('post.html', posts=[post], form=form,
                               comments=comments, pagination=pagination)
```

这个视图函数实例化了一个评论表单，并将其转入 post.html 模板，以便渲染。提交表单后，插入新评论的逻辑和处理博客文章的过程差不多。和 `Post` 模型一样，评论的 `author` 字段也不能直接设为 `current_user`，因为这个变量是上下文代理对象。真正的 `User` 对象要使用表达式 `current_user._get_current_object()` 获取。

评论按照时间戳顺序排列，新评论显示在列表的底部。提交评论后，请求结果是一个重定向，转回之前的 URL，但是在 `url_for()` 函数的参数中把 `page` 设为 `-1`，这是个特殊的页数，用来请求评论的最后一页，所以刚提交的评论才会出现在页面中。程序从查询字符串中获取页数，发现值为 `-1` 时，会计算评论的总量和总页数，得出真正要显示的页数。

文章的评论列表通过 `post.comments` 一对多关系获取，按照时间戳顺序进行排列，再使用与博客文章相同的技术分页显示。评论列表对象和分页对象都传入了模板，以便渲染。`FLASKY_COMMENTS_PER_PAGE` 配置变量也被加入 config.py 中，用来控制每页显示的评论数量。

评论的渲染过程在新模板 _comments.html 中进行，类似于 _posts.html，但使用的 CSS 类不同。_comments.html 模板要引入 post.html 中，放在文章正文下方，后面再显示分页导航。你可以在 GitHub 上的仓库中查看在这个程序里对模板所做的改动。

为了完善功能，我们还要在首页和资料页中加上指向评论页面的链接，如示例 13-5 所示。

示例 13-5　app/templates/_posts.html：链接到博客文章的评论

```
<a href="{{ url_for('.post', id=post.id) }}#comments">
    <span class="label label-primary">
        {{ post.comments.count() }} Comments
    </span>
</a>
```

注意链接文本中显示评论数量的方法。评论数量可以使用 SQLAlchemy 提供的 `count()` 过滤器轻易地从 `posts` 和 `comments` 表的一对多关系中获取。

指向评论页的链接结构也值得一说。这个链接的地址是在文章的固定链接后面加上一个 #comments 后缀。这个后缀称为 URL 片段，用于指定加载页面后滚动条所在的初始位置。Web 浏览器会寻找 `id` 等于 URL 片段的元素并滚动页面，让这个元素显示在窗口顶部。这个初始位置被设为 post.html 模板中评论区的标题，即 `<h4 id="comments">Comments<h4>`。显示有评论的页面如图 13-2 所示。

除此之外，分页导航所用的宏也要做些改动。评论的分页导航链接也要加上 #comments 片段，因此在 post.html 模板中调用宏时，传入片段参数。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 13a` 签出程序的这个版本。这个版本包含了一个数据库迁移，签出代码后记得要运行 `python manage.py db upgrade`。

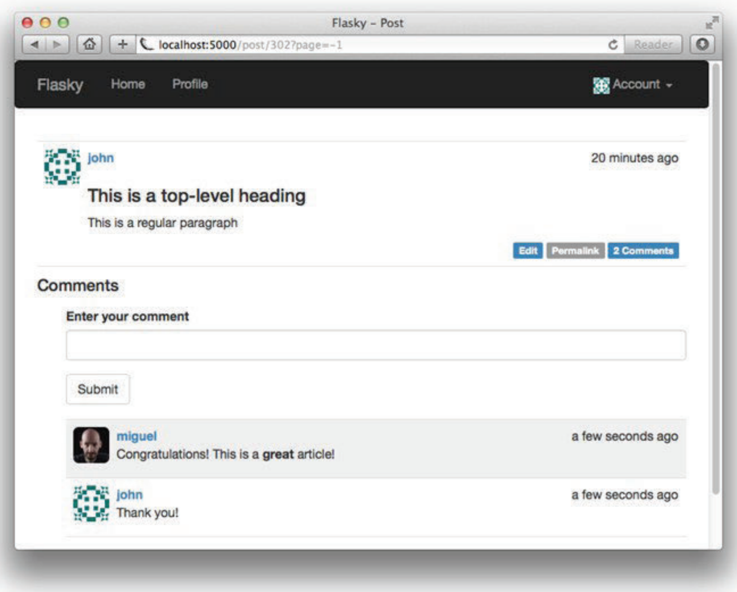


图 13-2　博客文章的评论

13.3　管理评论

我们在第 9 章定义了几个用户角色，它们分别具有不同的权限。其中一个权限是 `Permission.MODERATE_COMMENTS`，拥有此权限的用户可以管理其他用户的评论。

为了管理评论，我们要在导航条中添加一个链接，具有权限的用户才能看到。这个链接在 base.html 模板中使用条件语句添加，如示例 13-6 所示。

示例 13-6　app/templates/base.html：在导航条中加入管理评论链接

```
...
{% if current_user.can(Permission.MODERATE_COMMENTS) %}
<li><a href="{{ url_for('main.moderate') }}">Moderate Comments</a></li>
{% endif %}
...
```

管理页面在同一个列表中显示全部文章的评论，最近发布的评论会显示在前面。每篇评论的下方都会显示一个按钮，用来切换 `disabled` 属性的值。/moderate 路由的定义如示例 13-7 所示。

示例 13-7　app/main/views.py：管理评论的路由

```
@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate():
    page = request.args.get('page', 1, type=int)
    pagination = Comment.query.order_by(Comment.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    return render_template('moderate.html', comments=comments,
                           pagination=pagination, page=page)
```

这个函数很简单，它从数据库中读取一页评论，将其传入模板进行渲染。除了评论列表之外，还把分页对象和当前页数传入了模板。

moderate.html 模板也很简单，如示例 13-8 所示，因为它依靠之前创建的子模板 _comments.html 渲染评论。

示例 13-8　app/templates/moderate.html：评论管理页面的模板

```
{% extends "base.html" %}
{% import "_macros.html" as macros %}

{% block title %}Flasky - Comment Moderation{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Comment Moderation</h1>
</div>
```

```
{% set moderate = True %}
{% include '_comments.html' %}
{% if pagination %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.moderate') }}
</div>
{% endif %}
{% endblock %}
```

这个模板将渲染评论的工作交给 _comments.html 模板完成，但把控制权交给从属模板之前，会使用 Jinja2 提供的 `set` 指令定义一个模板变量 `moderate`，并将其值设为 `True`。这个变量用在 _comments.html 模板中，决定是否渲染评论管理功能。

_comments.html 模板中显示评论正文的部分要做两方面修改。对于普通用户（没设定 `moderate` 变量），不显示标记为有问题的评论。对于协管员（`moderate` 设为 `True`），不管评论是否被标记为有问题，都要显示，而且在正文下方还要显示一个用来切换状态的按钮。具体的改动如示例 13-9 所示。

示例 13-9 app/templates/_comments.html：渲染评论的正文

```
...
<div class="comment-body">
    {% if comment.disabled %}
    <p></p><i>This comment has been disabled by a moderator.</i></p>
    {% endif %}
    {% if moderate or not comment.disabled %}
        {% if comment.body_html %}
            {{ comment.body_html | safe }}
        {% else %}
            {{ comment.body }}
        {% endif %}
    {% endif %}
</div>
{% if moderate %}
    <br>
    {% if comment.disabled %}
    <a class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable',
        id=comment.id, page=page) }}">Enable</a>
    {% else %}
    <a class="btn btn-danger btn-xs" href="{{ url_for('.moderate_disable',
        id=comment.id, page=page) }}">Disable</a>
    {% endif %}
{% endif %}
...
```

做了上述改动之后，用户将看到一个关于有问题评论的简短提示。协管员既能看到这个提示，也能看到评论的正文。在每篇评论的下方，协管员还能看到一个按钮，用来切换评论的状态。点击按钮后会触发两个新路由中的一个，但具体触发哪一个取决于协管员要把评论设为什么状态。这两个新路由的定义如示例 13-10 所示。

示例 13-10　app/main/views.py：评论管理路由

```
@main.route('/moderate/enable/<int:id>')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate_enable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = False
    db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))

@main.route('/moderate/disable/<int:id>')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate_disable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = True
    db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))
```

上述启用路由和禁用路由先加载评论对象，把 disabled 字段设为正确的值，再把评论对象写入数据库。最后，重定向到评论管理页面（如图 13-3 所示），如果查询字符串中指定了 page 参数，会将其传入重定向操作。_comments.html 模板中的按钮指定了 page 参数，重定向后会返回之前的页面。

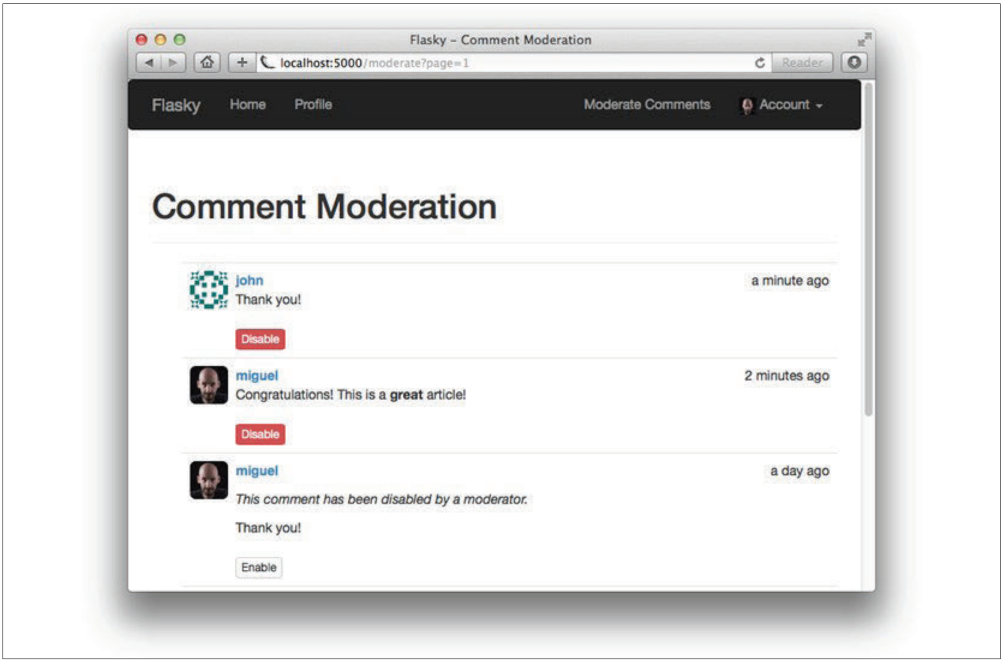


图 13-3　评论管理页面

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 13b` 签出程序的这个版本。

这一章结束了对社交功能的介绍。下一章，你将学到如何以 API 的形式开放程序的功能，从而让 Web 浏览器之外的客户端也能使用。

第14章

应用编程接口

最近几年，Web 程序有种趋势，那就是业务逻辑被越来越多地移到了客户端一侧，开创出了一种称为富互联网应用（Rich Internet Application，RIA）的架构。在 RIA 中，服务器的主要功能（有时是唯一功能）是为客户端提供数据存取服务。在这种模式中，服务器变成了 Web 服务或应用编程接口（Application Programming Interface，API）。

RIA 可采用多种协议与 Web 服务通信。远程过程调用（Remote Procedure Call，RPC）协议，例如 XML-RPC，及由其衍生的简单对象访问协议（Simplified Object Access Protocol，SOAP），在几年前比较受欢迎。最近，表现层状态转移（Representational State Transfer，REST）架构崭露头角，成为 Web 程序的新宠，因为这种架构建立在大家熟识的万维网基础之上。

Flask 是开发 REST 架构 Web 服务的理想框架，因为 Flask 天生轻量。在本章，你将学到如何使用 Flask 实现符合 REST 架构的 API。

14.1　REST简介

Roy Fielding 在其博士论文（http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm）中介绍了 Web 服务的 REST 架构方式，并列出了 6 个符合这一架构定义的特征。

客户端–服务器

客户端和服务器之间必须有明确的界线。

无状态

客户端发出的请求中必须包含所有必要的信息。服务器不能在两次请求之间保存客户端

的任何状态。

缓存

服务器发出的响应可以标记为可缓存或不可缓存，这样出于优化目的，客户端（或客户端和服务器之间的中间服务）可以使用缓存。

接口统一

客户端访问服务器资源时使用的协议必须一致，定义良好，且已经标准化。REST Web 服务最常使用的统一接口是 HTTP 协议。

系统分层

在客户端和服务器之间可以按需插入代理服务器、缓存或网关，以提高性能、稳定性和伸缩性。

按需代码

客户端可以选择从服务器上下载代码，在客户端的环境中执行。

14.1.1　资源就是一切

*资源*是 REST 架构方式的核心概念。在 REST 架构中，资源是程序中你要着重关注的事物。例如，在博客程序中，用户、博客文章和评论都是资源。

每个资源都要使用唯一的 URL 表示。还是以博客程序为例，一篇博客文章可以使用 URL /api/posts/12345 表示，其中 12345 是这篇文章的唯一标识符，使用文章在数据库中的主键表示。URL 的格式或内容无关紧要，只要资源的 URL 只表示唯一的一个资源即可。

某一类资源的集合也要有一个 URL。博客文章集合的 URL 可以是 /api/posts/，评论集合的 URL 可以是 /api/comments/。

API 还可以为某一类资源的逻辑子集定义集合 URL。例如，编号为 12345 的博客文章，其中的所有评论可以使用 URL /api/posts/12345/comments/ 表示。表示资源集合的 URL 习惯在末端加上一个斜线，代表一种“文件夹”结构。

> 注意，Flask 会特殊对待末端带有斜线的路由。如果客户端请求的 URL 的末端没有斜线，而唯一匹配的路由末端有斜线，Flask 会自动响应一个重定向，转向末端带斜线的 URL。反之则不会重定向。

14.1.2　请求方法

客户端程序在建立起的资源 URL 上发送请求，使用请求*方法*表示期望的操作。若要从博

客 API 中获取现有博客文章的列表，客户端可以向 http://www.exam-ple.com/api/posts/ 发送 `GET` 请求。若要插入一篇新博客文章，客户端可以向同一地址发送 `POST` 请求，而且请求主体中要包含博客文章的内容。若要获取编号为 12345 的博客文章，客户端可以向 http://www.example.com/api/posts/12345 发送 `GET` 请求。表 14-1 列出了 REST 架构 API 中常用的请求方法及其含义。

表14-1　REST架构API中使用的HTTP请求方法

请求方法	目　标	说　　明	HTTP状态码
`GET`	单个资源的 URL	获取目标资源	200
`GET`	资源集合的 URL	获取资源的集合（如果服务器实现了分页，就是一页中的资源）	200
`POST`	资源集合的 URL	创建新资源，并将其加入目标集合。服务器为新资源指派 URL，并在响应的 `Location` 首部中返回	201
`PUT`	单个资源的 URL	修改一个现有资源。如果客户端能为资源指派 URL，还可用来创建新资源	200
`DELETE`	单个资源的 URL	删除一个资源	200
`DELETE`	资源集合的 URL	删除目标集合中的所有资源	200

REST 架构不要求必须为一个资源实现所有的请求方法。如果资源不支持客户端使用的请求方法，响应的状态码为 405，返回“不允许使用的方法”。Flask 会自动处理这种错误。

14.1.3　请求和响应主体

在请求和响应的主体中，资源在客户端和服务器之间来回传送，但 REST 没有指定编码资源的方式。请求和响应中的 `Content-Type` 首部用于指明主体中资源的编码方式。使用 HTTP 协议中的内容协商机制，可以找到一种客户端和服务器都支持的编码方式。

REST Web 服务常用的两种编码方式是 JavaScript 对象表示法（JavaScript Object Notation，JSON）和可扩展标记语言（Extensible Markup Language，XML）。对基于 Web 的 RIA 来说，JSON 更具吸引力，因为 JSON 和 JavaScript 联系紧密，而 JavaScript 是 Web 浏览器使用的客户端脚本语言。继续以博客 API 为例，一篇博客文章对应的资源可以使用如下的 JSON 表示：

```
{
    "url": "http://www.example.com/api/posts/12345",
    "title": "Writing RESTful APIs in Python",
    "author": "http://www.example.com/api/users/2",
    "body": "... text of the article here ...",
    "comments": "http://www.example.com/api/posts/12345/comments"
}
```

注意，在这篇博客文章中，`url`、`author` 和 `comments` 字段都是完整的资源 URL。这是很重要的表示方法，因为客户端可以通过这些 URL 发掘新资源。

在设计良好的 REST API 中，客户端只需知道几个顶级资源的 URL，其他资源的 URL 则从响应中包含的链接上发掘。这就好比浏览网络时，你在自己知道的网页中点击链接发掘新网页。

14.1.4 版本

在传统的以服务器为中心的 Web 程序中，服务器完全掌控程序。更新程序时，只需在服务器上部署新版本就可更新所有的用户，因为运行在用户 Web 浏览器中的那部分程序也是从服务器上下载的。

但升级 RIA 和 Web 服务要复杂得多，因为客户端程序和服务器上的程序是独立开发的，有时甚至由不同的人进行开发。你可以考虑一下这种情况，即一个程序的 REST Web 服务被很多客户端使用，其中包括 Web 浏览器和智能手机原生应用。服务器可以随时更新 Web 浏览器中的客户端，但无法强制更新智能手机中的应用，更新前先要获得机主的许可。即便机主想进行更新，也不能保证新版应用上传到所有应用商店的时机都完全吻合新服务器端版本的部署。

基于以上原因，Web 服务的容错能力要比一般的 Web 程序强，而且还要保证旧版客户端能继续使用。这一问题的常见解决办法是使用*版本*区分 Web 服务所处理的 URL。例如，首次发布的博客 Web 服务可以通过 /api/v1.0/posts/ 提供博客文章的集合。

在 URL 中加入 Web 服务的版本有助于条理化管理新旧功能，让服务器能为新客户端提供新功能，同时继续支持旧版客户端。博客服务可能会修改博客文章使用的 JSON 格式，同时通过 /api/v1.1/posts/ 提供修改后的博客文章，而客户端仍能通过 /api/v1.0/posts/ 获取旧的 JSON 格式。在一段时间内，服务器要同时处理 v1.1 和 v1.0 这两个版本的 URL。

提供多版本支持会增加服务器的维护负担，但在某些情况下，这是不破坏现有部署且能让程序不断发展的唯一方式。

14.2 使用Flask提供REST Web服务

使用 Flask 创建 REST Web 服务很简单。使用熟悉的 `route()` 修饰器及其 `methods` 可选参数可以声明服务所提供资源 URL 的路由。处理 JSON 数据同样简单，因为请求中包含的 JSON 数据可通过 `request.json` 这个 Python 字典获取，并且需要包含 JSON 的响应可以使用 Flask 提供的辅助函数 `jsonify()` 从 Python 字典中生成。

以下几节将介绍如何扩展 Flasky，创建一个 REST Web 服务，以便让客户端访问博客文章及相关资源。

14.2.1 创建API蓝本

REST API 相关的路由是一个自成一体的程序子集，所以为了更好地组织代码，我们最好把这些路由放到独立的蓝本中。这个程序 API 蓝本的基本结构如示例 14-1 所示。

示例 14-1 API 蓝本的结构

```
|-flasky
  |-app/
    |-api_1_0
      |-__init__.py
      |-users.py
      |-posts.py
      |-comments.py
      |-authentication.py
      |-errors.py
      |-decorators.py
```

注意，API 包的名字中有一个版本号。如果需要创建一个向前兼容的 API 版本，可以添加一个版本号不同的包，让程序同时支持两个版本的 API。

在这个 API 蓝本中，各资源分别在不同的模块中实现。蓝本中还包含处理认证、错误以及提供自定义修饰器的模块。蓝本的构造文件如示例 14-2 所示。

示例 14-2 app/api_1_0/__init__.py：API 蓝本的构造文件

```
from flask import Blueprint

api = Blueprint('api', __name__)

from . import authentication, posts, users, comments, errors
```

注册 API 蓝本的代码如示例 14-3 所示。

示例 14-3 app/_init_.py：注册 API 蓝本

```
def create_app(config_name):
    # ...
    from .api_1_0 import api as api_1_0_blueprint
    app.register_blueprint(api_1_0_blueprint, url_prefix='/api/v1.0')
    # ...
```

14.2.2 错误处理

REST Web 服务将请求的状态告知客户端时，会在响应中发送适当的 HTTP 状态码，并将额外信息放入响应主体。客户端能从 Web 服务得到的常见状态码如表 14-2 所示。

表14-2　API返回的常见HTTP状态码

HTTP状态码	名　　称	说　　明
200	OK（成功）	请求成功完成
201	Created（已创建）	请求成功完成并创建了一个新资源
400	Bad request（坏请求）	请求不可用或不一致
401	Unauthorized（未授权）	请求未包含认证信息
403	Forbidden（禁止）	请求中发送的认证密令无权访问目标
404	Notfound（未找到）	URL 对应的资源不存在
405	Method not allowed（不允许使用的方法）	指定资源不支持请求使用的方法
500	Internal server error（内部服务器错误）	处理请求的过程中发生意外错误

处理 404 和 500 状态码时会有点小麻烦，因为这两个错误是由 Flask 自己生成的，而且一般会返回 HTML 响应，这很可能会让 API 客户端困惑。

为所有客户端生成适当响应的一种方法是，在错误处理程序中根据客户端请求的格式改写响应，这种技术称为*内容协商*。示例 14-4 是改进后的 404 错误处理程序，它向 Web 服务客户端发送 JSON 格式响应，除此之外都发送 HTML 格式响应。500 错误处理程序的写法类似。

示例 14-4　app/main/errors.py：使用 HTTP 内容协商处理错误

```
@main.app_errorhandler(404)
def page_not_found(e):
    if request.accept_mimetypes.accept_json and \
            not request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'not found'})
        response.status_code = 404
        return response
    return render_template('404.html'), 404
```

这个新版错误处理程序检查 `Accept` 请求首部（Werkzeug 将其解码为 `request.accept_mimetypes`），根据首部的值决定客户端期望接收的响应格式。浏览器一般不限制响应的格式，所以只为接受 JSON 格式而不接受 HTML 格式的客户端生成 JSON 格式响应。

其他状态码都由 Web 服务生成，因此可在蓝本的 errors.py 模块作为辅助函数实现。示例 14-5 是 403 错误的处理程序，其他错误处理程序的写法类似。

示例 14-5　app/api_1_0/errors.py：API 蓝本中 403 状态码的错误处理程序

```
def forbidden(message):
    response = jsonify({'error': 'forbidden', 'message': message})
    response.status_code = 403
    return response
```

现在，Web 服务的视图函数可以调用这些辅助函数生成错误响应了。

14.2.3　使用Flask-HTTPAuth认证用户

和普通的 Web 程序一样，Web 服务也需要保护信息，确保未经授权的用户无法访问。为此，RIA 必须询问用户的登录密令，并将其传给服务器进行验证。

我们前面说过，REST Web 服务的特征之一是*无状态*，即服务器在两次请求之间不能“记住”客户端的任何信息。客户端必须在发出的请求中包含所有必要信息，因此所有请求都必须包含用户密令。

程序当前的登录功能是在 Flask-Login 的帮助下实现的，可以把数据存储在用户会话中。默认情况下，Flask 把会话保存在客户端 cookie 中，因此服务器没有保存任何用户相关信息，都转交给客户端保存。这种实现方式看起来遵守了 REST 架构的无状态要求，但在 REST Web 服务中使用 cookie 有点不现实，因为 Web 浏览器之外的客户端很难提供对 cookie 的支持。鉴于此，使用 cookie 并不是一个很好的设计选择。

> REST 架构的无状态要求看起来似乎过于严格，但这并不是随意提出的要求，无状态的服务器伸缩起来更加简单。如果服务器保存了客户端的相关信息，就必须提供一个所有服务器都能访问的共享缓存，这样才能保证一直使用同一台服务器处理特定客户端的请求。这样的需求很难实现。

因为 REST 架构基于 HTTP 协议，所以发送密令的最佳方式是使用 HTTP *认证*，基本认证和摘要认证都可以。在 HTTP 认证中，用户密令包含在请求的 `Authorization` 首部中。

HTTP 认证协议很简单，可以直接实现，不过 Flask-HTTPAuth 扩展提供了一个便利的包装，可以把协议的细节隐藏在修饰器之中，类似于 Flask-Login 提供的 `login_required` 修饰器。

Flask-HTTPAuth 使用 `pip` 安装：

```
(venv) $ pip install flask-httpauth
```

在将 HTTP 基本认证的扩展进行初始化之前，我们先要创建一个 `HTTPBasicAuth` 类对象。和 Flask-Login 一样，Flask-HTTPAuth 不对验证用户密令所需的步骤做任何假设，因此所需的信息在回调函数中提供。示例 14-6 展示了如何初始化 Flask-HTTPAuth 扩展，以及如何在回调函数中验证密令。

示例 14-6　app/api_1_0/authentication.py：初始化 Flask-HTTPAuth

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.verify_password
def verify_password(email, password):
```

```
    if email == '':
        g.current_user = AnonymousUser()
        return True
    user = User.query.filter_by(email = email).first()
    if not user:
        return False
    g.current_user = user
    return user.verify_password(password)
```

由于这种用户认证方法只在 API 蓝本中使用，所以 Flask-HTTPAuth 扩展只在蓝本包中初始化，而不像其他扩展那样要在程序包中初始化。

电子邮件和密码使用 `User` 模型中现有的方法验证。如果登录密令正确，这个验证回调函数就返回 `True`，否则返回 `False`。API 蓝本也支持匿名用户访问，此时客户端发送的电子邮件字段必须为空。

验证回调函数把通过认证的用户保存在 Flask 的全局对象 `g` 中，如此一来，视图函数便能进行访问。注意，匿名登录时，这个函数返回 `True` 并把 Flask-Login 提供的 `AnonymousUser` 类实例赋值给 `g.current_user`。

> 由于每次请求时都要传送用户密令，所以 API 路由最好通过安全的 HTTP 提供，加密所有的请求和响应。

如果认证密令不正确，服务器向客户端返回 401 错误。默认情况下，Flask-HTTPAuth 自动生成这个状态码，但为了和 API 返回的其他错误保持一致，我们可以自定义这个错误响应，如示例 14-7 所示。

示例 14-7 app/api_1_0/authentication.py：Flask-HTTPAuth 错误处理程序

```
@auth.error_handler
def auth_error():
    return unauthorized('Invalid credentials')
```

为保护路由，可使用修饰器 `auth.login_required`：

```
@api.route('/posts/')
@auth.login_required
def get_posts():
    pass
```

不过，这个蓝本中的所有路由都要使用相同的方式进行保护，所以我们可以在 `before_request` 处理程序中使用一次 `login_required` 修饰器，应用到整个蓝本，如示例 14-8 所示。

示例 14-8 app/api_1_0/authentication.py：在 before_request 处理程序中进行认证

```
from .errors import forbidden_error
```

```
@api.before_request
@auth.login_required
def before_request():
    if not g.current_user.is_anonymous and \
            not g.current_user.confirmed:
        return forbidden('Unconfirmed account')
```

现在，API 蓝本中的所有路由都能进行自动认证。而且作为附加认证，before_request 处理程序还会拒绝已通过认证但没有确认账户的用户。

14.2.4　基于令牌的认证

每次请求时，客户端都要发送认证密令。为了避免总是发送敏感信息，我们可以提供一种基于令牌的认证方案。

使用基于令牌的认证方案时，客户端要先把登录密令发送给一个特殊的 URL，从而生成认证令牌。一旦客户端获得令牌，就可用令牌代替登录密令认证请求。出于安全考虑，令牌有过期时间。令牌过期后，客户端必须重新发送登录密令以生成新令牌。令牌落入他人之手所带来的安全隐患受限于令牌的短暂使用期限。为了生成和验证认证令牌，我们要在 User 模型中定义两个新方法。这两个新方法用到了 itsdangerous 包，如示例 14-9 所示。

示例 14-9　app/models.py：支持基于令牌的认证

```
class User(db.Model):
    # ...
    def generate_auth_token(self, expiration):
        s = Serializer(current_app.config['SECRET_KEY'],
                       expires_in=expiration)
        return s.dumps({'id': self.id})

    @staticmethod
    def verify_auth_token(token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return None
        return User.query.get(data['id'])
```

generate_auth_token() 方法使用编码后的用户 id 字段值生成一个签名令牌，还指定了以秒为单位的过期时间。verify_auth_token() 方法接受的参数是一个令牌，如果令牌可用就返回对应的用户。verify_auth_token() 是静态方法，因为只有解码令牌后才能知道用户是谁。

为了能够认证包含令牌的请求，我们必须修改 Flask-HTTPAuth 提供的 verify_password 回调，除了普通的密令之外，还要接受令牌。修改后的回调如示例 14-10 所示。

示例 14-10　app/api_1_0/authentication.py：支持令牌的改进验证回调

```
@auth.verify_password
def verify_password(email_or_token, password):
    if email_or_token == '':
        g.current_user = AnonymousUser()
        return True
    if password == '':
        g.current_user = User.verify_auth_token(email_or_token)
        g.token_used = True
        return g.current_user is not None
    user = User.query.filter_by(email=email_or_token).first()
    if not user:
        return False
    g.current_user = user
    g.token_used = False
    return user.verify_password(password)
```

在这个新版本中，第一个认证参数可以是电子邮件地址或认证令牌。如果这个参数为空，那就和之前一样，假定是匿名用户。如果密码为空，那就假定 `email_or_token` 参数提供的是令牌，按照令牌的方式进行认证。如果两个参数都不为空，假定使用常规的邮件地址和密码进行认证。在这种实现方式中，基于令牌的认证是可选的，由客户端决定是否使用。为了让视图函数能区分这两种认证方法，我们添加了 `g.token_used` 变量。

把认证令牌发送给客户端的路由也要添加到 API 蓝本中，具体实现如示例 14-11 所示。

示例 14-11　app/api_1_0/authentication.py：生成认证令牌

```
@api.route('/token')
def get_token():
    if g.current_user.is_anonymous() or g.token_used:
        return unauthorized('Invalid credentials')
    return jsonify({'token': g.current_user.generate_auth_token(
        expiration=3600), 'expiration': 3600})
```

由于这个路由也在蓝本中，所以添加到 `before_request` 处理程序上的认证机制也会用在这个路由上。为了避免客户端使用旧令牌申请新令牌，要在视图函数中检查 `g.token_used` 变量的值，如果使用令牌进行认证就拒绝请求。这个视图函数返回 JSON 格式的响应，其中包含了过期时间为 1 小时的令牌。JSON 格式的响应也包含过期时间。

14.2.5　资源和JSON的序列化转换

开发 Web 程序时，经常需要在资源的内部表示和 JSON 之间进行转换。JSON 是 HTTP 请求和响应使用的传输格式。示例 14-12 是新添加到 `Post` 类中的 `to_json()` 方法。

示例 14-12　app/models.py：把文章转换成 JSON 格式的序列化字典

```
class Post(db.Model):
    # ...
    def to_json(self):
```

```
        json_post = {
            'url': url_for('api.get_post', id=self.id, _external=True),
            'body': self.body,
            'body_html': self.body_html,
            'timestamp': self.timestamp,
            'author': url_for('api.get_user', id=self.author_id,
                              _external=True),
            'comments': url_for('api.get_post_comments', id=self.id,
                                _external=True)
            'comment_count': self.comments.count()
        }
        return json_post
```

url、author 和 comments 字段要分别返回各自资源的 URL，因此它们使用 url_for() 生成，所调用的路由即将在 API 蓝本中定义。注意，所有 url_for() 方法都指定了参数 _external=True，这么做是为了生成完整的 URL，而不是生成传统 Web 程序中经常使用的相对 URL。

这段代码还说明表示资源时可以使用虚构的属性。comment_count 字段是博客文章的评论数量，并不是模型的真实属性，它之所以包含在这个资源中是为了便于客户端使用。

User 模型的 to_json() 方法可以按照 Post 模型的方式定义，如示例 14-13 所示。

示例 14-13　app/models.py：把用户转换成 JSON 格式的序列化字典

```
class User(UserMixin, db.Model):
    # ...
    def to_json(self):
        json_user = {
            'url': url_for('api.get_post', id=self.id, _external=True),
            'username': self.username,
            'member_since': self.member_since,
            'last_seen': self.last_seen,
            'posts': url_for('api.get_user_posts', id=self.id, _external=True),
            'followed_posts': url_for('api.get_user_followed_posts',
                                      id=self.id, _external=True),
            'post_count': self.posts.count()
        }
        return json_user
```

注意，为了保护隐私，这个方法中用户的某些属性没有加入响应，例如 email 和 role。这段代码再次说明，提供给客户端的资源表示没必要和数据库模型的内部表示完全一致。

把 JSON 转换成模型时面临的问题是，客户端提供的数据可能无效、错误或者多余。示例 14-14 是从 JSON 格式数据创建 Post 模型实例的方法。

示例 14-14　app/models.py：从 JSON 格式数据创建一篇博客文章

```
from app.exceptions import ValidationError

class Post(db.Model):
```

```
    # ...
    @staticmethod
    def from_json(json_post):
        body = json_post.get('body')
        if body is None or body == '':
            raise ValidationError('post does not have a body')
        return Post(body=body)
```

如你所见，上述代码在实现过程中只选择使用 JSON 字典中的 `body` 属性，而把 `body_html` 属性忽略了，因为只要 `body` 属性的值发生变化，就会触发一个 SQLAlchemy 事件，自动在服务器端渲染 Markdown。除非允许客户端倒填日期（这个程序并不提供此功能），否则无需指定 `timestamp` 属性。由于客户端无权选择博客文章的作者，所以没有使用 `author` 字段。`author` 字段唯一能使用的值是通过认证的用户。`comments` 和 `comment_count` 属性使用数据库关系自动生成，因此其中没有创建模型所需的有用信息。最后，`url` 字段也被忽略了，因为在这个实现中资源的 URL 由服务器指派，而不是客户端。

注意如何检查错误。如果没有 `body` 字段或者其值为空，`from_json()` 方法会抛出 `ValidationError` 异常。在这种情况下，抛出异常才是处理错误的正确方式，因为 `from_json()` 方法并没有掌握处理问题的足够信息，唯有把错误交给调用者，由上层代码处理这个错误。`ValidationError` 类是 Python 中 `ValueError` 类的简单子类，具体定义如示例 14-15 所示。

示例 14-15 app/exceptions.py：ValidationError 异常

```
class ValidationError(ValueError):
    pass
```

现在，程序需要向客户端提供适当的响应以处理这个异常。为了避免在视图函数中编写捕获异常的代码，我们可创建一个全局异常处理程序。对于 `ValidationError` 异常，其处理程序如示例 14-16 所示。

示例 14-16 app/api_1_0/errors.py：API 中 ValidationError 异常的处理程序

```
@api.errorhandler(ValidationError)
def validation_error(e):
    return bad_request(e.args[0])
```

这里使用的 `errorhandler` 修饰器和注册 HTTP 状态码处理程序时使用的是同一个，只不过此时接收的参数是 `Exception` 类，只要抛出了指定类的异常，就会调用被修饰的函数。注意，这个修饰器从 API 蓝本中调用，所以只有当处理蓝本中的路由时抛出了异常才会调用这个处理程序。

使用这个技术时，视图函数中得代码可以写得十分简洁明，而且无需检查错误。例如：

```
@api.route('/posts/', methods=['POST'])
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
```

```
        db.session.add(post)
        db.session.commit()
        return jsonify(post.to_json())
```

14.2.6　实现资源端点

现在我们需要实现用于处理不同资源的路由。`GET` 请求往往是最简单的，因为它们只返回信息，无需修改信息。示例 14-17 是博客文章的两个 `GET` 请求处理程序。

示例 14-17　app/api_1_0/posts.py：文章资源 GET 请求的处理程序

```
@api.route('/posts/')
@auth.login_required
def get_posts():
    posts = Post.query.all()
    return jsonify({ 'posts': [post.to_json() for post in posts] })

@api.route('/posts/<int:id>')
@auth.login_required
def get_post(id):
    post = Post.query.get_or_404(id)
    return jsonify(post.to_json())
```

第一个路由处理获取文章集合的请求。这个函数使用列表推导生成所有文章的 JSON 版本。第二个路由返回单篇博客文章，如果在数据库中没找到指定 `id` 对应的文章，则返回 404 错误。

> 404 错误的处理程序在程序层定义，如果客户端请求 JSON 格式，就要返回 JSON 格式响应。如果要根据 Web 服务定制响应内容，也可在 API 蓝本中重新定义 404 错误处理程序。

博客文章资源的 `POST` 请求处理程序把一篇新博客文章插入数据库。路由的定义如示例 14-18 所示。

示例 14-18　app/api_1_0/posts.py：文章资源 POST 请求的处理程序

```
@api.route('/posts/', methods=['POST'])
@permission_required(Permission.WRITE_ARTICLES)
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json()), 201, \
        {'Location': url_for('api.get_post', id=post.id, _external=True)}
```

这个视图函数包含在 `permission_required` 修饰器（下面的示例中会定义）中，确保通过

认证的用户有写博客文章的权限。得益于前面实现的错误处理程序，创建博客文章的过程变得很直观。博客文章从 JSON 数据中创建，其作者就是通过认证的用户。这个模型写入数据库之后，会返回 201 状态码，并把 Location 首部的值设为刚创建的这个资源的 URL。

注意，为便于客户端操作，响应的主体中包含了新建的资源。如此一来，客户端就无需在创建资源后再立即发起一个 GET 请求以获取资源。

用来防止未授权用户创建新博客文章的 permission_required 修饰器和程序中使用的类似，但会针对 API 蓝本进行自定义。具体实现如示例 14-19 所示。

示例 14-19 app/api_1_0/decorators.py：permission_required 修饰器

```
def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not g.current_user.can(permission):
                return forbidden('Insufficient permissions')
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

博客文章 PUT 请求的处理程序用来更新现有资源，如示例 14-20 所示。

示例 14-20 app/api_1_0/posts.py：文章资源 PUT 请求的处理程序

```
@api.route('/posts/<int:id>', methods=['PUT'])
@permission_required(Permission.WRITE_ARTICLES)
def edit_post(id):
    post = Post.query.get_or_404(id)
    if g.current_user != post.author and \
            not g.current_user.can(Permission.ADMINISTER):
        return forbidden('Insufficient permissions')
    post.body = request.json.get('body', post.body)
    db.session.add(post)
    return jsonify(post.to_json())
```

本例中要进行的权限检查更为复杂。修饰器用来检查用户是否有写博客文章的权限，但为了确保用户能编辑博客文章，这个函数还要保证用户是文章的作者或者是管理员。这个检查直接添加到视图函数中。如果这种检查要应用于多个视图函数，为避免代码重复，最好的方法是为其创建修饰器。

因为程序不允许删除文章，所以没必要实现 DELETE 请求方法的处理程序。

用户资源和评论资源的处理程序实现方式类似。表 14-3 列出了这个程序要实现的资源。你可到 GitHub 仓库（https://github.com/miguelgrinberg/flasky）中获取完整的实现，以便学习研究。

表14-3　Flasky API资源

资源URL	方　　法	说　　明
/users/<int:id>	GET	一个用户
/users/<int:id>/posts/	GET	一个用户发布的博客文章
/users/<int:id>/timeline/	GET	一个用户所关注用户发布的文章
/posts/	GET、POST	所有博客文章
/posts/<int:id>	GET、PUT	一篇博客文章
/posts/<int:id/>comments/	GET、POST	一篇博客文章中的评论
/comments/	GET	所有评论
/comments/<int:id>	GET	一篇评论

注意，这些资源只允许客户端实现 Web 程序提供的部分功能。支持的资源可以按需扩展，比如说提供关注者资源、支持评论管理，以及实现客户端需要的其他功能。

14.2.7　分页大型资源集合

对大型资源集合来说，获取集合的 GET 请求消耗很大，而且难以管理。和 Web 程序一样，Web 服务也可以对集合进行分页。

示例 14-21 是分页博客文章列表的一种实现方式。

示例 14-21　app/api_1_0/posts.py：分页文章资源

```
@api.route('/posts/')
def get_posts():
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    prev = None
    if pagination.has_prev:
        prev = url_for('api.get_posts', page=page-1, _external=True)
    next = None
    if pagination.has_next:
        next = url_for('api.get_posts', page=page+1, _external=True)
    return jsonify({
        'posts': [post.to_json() for post in posts],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })
```

JSON 格式响应中的 posts 字段依旧包含各篇文章，但现在这只是完整集合的一部分。如果资源有上一页和下一页，prev 和 next 字段分别表示上一页和下一页资源的 URL。count 是集合中博客文章的总数。

这种技术可应用于所有返回集合的路由。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 14a` 签出程序的这个版本。为保证安装了所有依赖，你还要运行 `pip install -r requirements/dev.txt`。

14.2.8　使用HTTPie测试Web服务

测试 Web 服务时必须使用 HTTP 客户端。最常使用的两个在命令行中测试 Web 服务的客户端是 curl 和 HTTPie。后者的命令行更简洁，可读性也更高。HTTPie 使用 `pip` 安装：

```
(venv) $ pip install httpie
```

`GET` 请求可按照如下的方式发起：

```
(venv) $ http --json --auth <email>:<password> GET \
> http://127.0.0.1:5000/api/v1.0/posts
HTTP/1.0 200 OK
Content-Length: 7018
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:11:24 GMT
Server: Werkzeug/0.9.4 Python/2.7.3

{
    "posts": [
        ...
    ],
    "prev": null
    "next": "http://127.0.0.1:5000/api/v1.0/posts/?page=2",
    "count": 150
}
```

注意响应中的分页链接。因为这是第一页，所以没有上一页，不过返回了获取下一页的 URL 和总数。

匿名用户可发送空邮件地址和密码以发起相同的请求：

```
(venv) $ http --json --auth : GET http://127.0.0.1:5000/api/v1.0/posts/
```

下面这个命令发送 `POST` 请求以添加一篇新博客文章：

```
(venv) $ http --auth <email>:<password> --json POST \
> http://127.0.0.1:5000/api/v1.0/posts/ \
> "body=I'm adding a post from the *command line*."
HTTP/1.0 201 CREATED
Content-Length: 360
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:30:27 GMT
```

```
Location: http://127.0.0.1:5000/api/v1.0/posts/111
Server: Werkzeug/0.9.4 Python/2.7.3

{
    "author": "http://127.0.0.1:5000/api/v1.0/users/1",
    "body": "I'm adding a post from the *command line*.",
    "body_html": "<p>I'm adding a post from the <em>command line</em>.</p>",
    "comments": "http://127.0.0.1:5000/api/v1.0/posts/111/comments",
    "comment_count": 0,
    "timestamp": "Sun, 22 Dec 2013 08:30:27 GMT",
    "url": "http://127.0.0.1:5000/api/v1.0/posts/111"
}
```

要想使用认证令牌，可向 /api/v1.0/token 发送请求：

```
(venv) $ http --auth <email>:<password> --json GET \
> http://127.0.0.1:5000/api/v1.0/token
HTTP/1.0 200 OK
Content-Length: 162
Content-Type: application/json
Date: Sat, 04 Jan 2014 08:38:47 GMT
Server: Werkzeug/0.9.4 Python/3.3.3

{
    "expiration": 3600,
    "token": "eyJpYXQiOjEzODg4MjQ3MjcsImV4cCI6MTM4ODgyODMyNywiYWxnIjoiSFMy..."
}
```

在接下来的 1 小时中，这个令牌可用于访问 API，请求时要和空密码一起发送：

```
(venv) $ http --json --auth eyJpYXQ...: GET http://127.0.0.1:5000/api/v1.0/posts/
```

令牌过期后，请求会返回 401 错误，表示需要获取新令牌。

祝贺你！我们在这一章结束了第二部分，至此，Flasky 的功能开发阶段就完全结束了。很显然，下一步我们要部署 Flasky。在部署过程中，我们会遇到新的挑战，这就是第三部分的主题。

第三部分

成功在望

第 15 章

测试

编写单元测试主要有两个目的。实现新功能时，单元测试能够确保新添加的代码按预期方式运行。当然，这个过程也可手动完成，不过自动化测试显然能有效节省时间和精力。

另外，一个更重要的目的是，每次修改程序后，运行单元测试能保证现有代码的功能没有退化。也就是说，改动没有影响原有代码的正常运行。

在最开始，单元测试就是 Flasky 开发的一部分，我们为数据库模型类中实现的程序功能编写了测试。模型类很容易在运行中的程序上下文之外进行测试，因此不用花费太多精力，为数据库模型中实现的全部功能编写单元测试，这至少能有效保证程序这部分在不断完善的过程中仍能按预期运行。

在本章，我们将讨论如何改进、增强单元测试。

15.1　获取代码覆盖报告

编写测试组件很重要，但知道测试的好坏同样重要。代码覆盖工具用来统计单元测试检查了多少程序功能，并提供一个详细的报告，说明程序的哪些代码没有测试到。这个信息非常重要，因为它能指引你为最需要测试的部分编写新测试。

Python 提供了一个优秀的代码覆盖工具，称为 coverage，你可以使用 `pip` 进行安装：

```
(venv) $ pip install coverage
```

这个工具本身是一个命令行脚本，可在任何一个 Python 程序中检查代码覆盖。除此之外，

它还提供了更方便的脚本访问功能，使用编程方式启动覆盖检查引擎。为了能更好地把覆盖检测集成到启动脚本 manage.py 中，我们可以增强第 7 章中自定义的 `test` 命令，添加可选选项 `--coverage`。这个选项的实现方式如示例 15-1 所示。

示例 15-1 manage.py：覆盖检测

```
#!/usr/bin/env python
import os
COV = None
if os.environ.get('FLASK_COVERAGE'):
    import coverage
    COV = coverage.coverage(branch=True, include='app/*')
    COV.start()

# ...

@manager.command
def test(coverage=False):
    """Run the unit tests."""
    if coverage and not os.environ.get('FLASK_COVERAGE'):
        import sys
        os.environ['FLASK_COVERAGE'] = '1'
        os.execvp(sys.executable, [sys.executable] + sys.argv)
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
    if COV:
        COV.stop()
        COV.save()
        print('Coverage Summary:')
        COV.report()
        basedir = os.path.abspath(os.path.dirname(__file__))
        covdir = os.path.join(basedir, 'tmp/coverage')
        COV.html_report(directory=covdir)
        print('HTML version: file://%s/index.html' % covdir)
        COV.erase()

# ...
```

在 Flask-Script 中，自定义命令很简单。若想为 `test` 命令添加一个布尔值选项，只需在 `test()` 函数中添加一个布尔值参数即可。Flask-Script 根据参数名确定选项名，并据此向函数中传入 `True` 或 `False`。

不过，把代码覆盖集成到 manage.py 脚本中有个小问题。`test()` 函数收到 `--coverage` 选项的值后再启动覆盖检测已经晚了，那时全局作用域中的所有代码都已经执行了。为了检测的准确性，设定完环境变量 `FLASK_COVERAGE` 后，脚本会重启。再次运行时，脚本顶端的代码发现已经设定了环境变量，于是立即启动覆盖检测。

函数 `coverage.coverage()` 用于启动覆盖检测引擎。`branch=True` 选项开启分支覆盖分析，除了跟踪哪行代码已经执行外，还要检查每个条件语句的 `True` 分支和 `False` 分支是否都执

行了。include 选项用来限制程序包中文件的分析范围，只对这些文件中的代码进行覆盖检测。如果不指定 include 选项，虚拟环境中安装的全部扩展和测试代码都会包含进覆盖报告中，给报告添加很多杂项。

执行完所有测试后，text() 函数会在终端输出报告，同时还会生成一个使用 HTML 编写的精美报告并写入硬盘。HTML 格式的报告非常适合直观形象地展示覆盖信息，因为它按照源码的使用情况给代码行加上了不同的颜色。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 git checkout 15a 签出程序的这个版本。为保证安装了所有依赖，你还要运行 pip install -r requirements/dev.txt。

文本格式的报告示例如下：

```
(venv) $ python manage.py test --coverage
...
.----------------------------------------------------------------------
Ran 19 tests in 50.609s

OK
Coverage Summary:
Name                          Stmts   Miss Branch BrMiss  Cover   Missing
...
.-----------------------------------------------------------------------
app/__init__                     33      0      0      0   100%
app/api_1_0/__init__              3      0      0      0   100%
app/api_1_0/authentication       30     19     11     11    27%
app/api_1_0/comments             40     30     12     12    19%
app/api_1_0/decorators           11      3      2      2    62%
app/api_1_0/errors               17     10      0      0    41%
app/api_1_0/posts                35     23      9      9    27%
app/api_1_0/users                30     24     12     12    14%
app/auth/__init__                 3      0      0      0   100%
app/auth/forms                   45      8      8      8    70%
app/auth/views                  109     84     41     41    17%
app/decorators                   14      3      2      2    69%
app/email                        15      9      0      0    40%
app/exceptions                    2      0      0      0   100%
app/main/__init__                 6      1      0      0    83%
app/main/errors                  20     15      9      9    17%
app/main/forms                   39      7      8      8    68%
app/main/views                  169    131     36     36    19%
app/models                      243     62     44     17    72%
.-----------------------------------------------------------------------
TOTAL                           864    429    194    167    44%
HTML version: file:///home/flask/flasky/tmp/coverage/index.html
```

上述报告显示，整体覆盖率为 44%。情况并不遭，但也不太好。现阶段，模型类是单元

测试的关注焦点，它共包含 243 个语句，测试覆盖了其中 72% 的语句。很明显，`main` 和 `auth` 蓝本中的 views.py 文件以及 `api_1_0` 蓝本中的路由的覆盖率都很低，因为我们没有为这些代码编写单元测试。

有了这个报告，我们就能很容易确定向测试组件中添加哪些测试以提高覆盖率。但遗憾的是，并非程序的所有组成部分都像数据库模型那样易于测试。在接下来的两节，我们将介绍更高级的测试策略，可用于测试视图函数、表单和模板。

注意，出于排版考虑，上述示例报告省略了“Missing”列的内容。这一列显示测试没有覆盖的源码行，是一个由行号范围组成的长列表。

15.2　Flask测试客户端

程序的某些代码严重依赖运行中的程序所创建的环境。例如，你不能直接调用视图函数中的代码进行测试，因为这个函数可能需要访问 Flask 上下文全局变量，如 `request` 或 `session`；视图函数可能还等待接收 `POST` 请求中的表单数据，而且某些视图函数要求用户先登录。简而言之，视图函数只能在请求上下文和运行中的程序里运行。

Flask 内建了一个*测试客户端*用于解决（至少部分解决）这一问题。测试客户端能复现程序运行在 Web 服务器中的环境，让测试扮演成客户端从而发送请求。

在测试客户端中运行的视图函数和正常情况下的没有太大区别，服务器收到请求，将其分配给适当的视图函数，视图函数生成响应，将其返回给测试客户端。执行视图函数后，生成的响应会传入测试，检查是否正确。

15.2.1　测试Web程序

示例 15-2 是一个使用测试客户端编写的单元测试框架。

示例 15-2　tests/test_client.py：使用 Flask 测试客户端编写的测试框架

```
import unittest
from app import create_app, db
from app.models import User, Role

class FlaskClientTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()
        Role.insert_roles()
        self.client = self.app.test_client(use_cookies=True)

    def tearDown(self):
```

```
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_home_page(self):
        response = self.client.get(url_for('main.index'))
        self.assertTrue('Stranger' in response.get_data(as_text=True))
```

测试用例中的实例变量 `self.client` 是 Flask 测试客户端对象。在这个对象上可调用方法向程序发起请求。如果创建测试客户端时启用了 `use_cookies` 选项，这个测试客户端就能像浏览器一样接收和发送 cookie，因此能使用依赖 cookie 的功能记住请求之间的上下文。值得一提的是，这个选项可用来启用用户会话，让用户登录和退出。

`test_home_page()` 测试作为一个简单的例子演示了测试客户端的作用。在这个例子中，客户端向首页发起了一个请求。在测试客户端上调用 `get()` 方法得到的结果是一个 Flask`Response` 对象，内容是调用视图函数得到的响应。为了检查测试是否成功，要在响应主体中搜索是否包含 `"Stranger"` 这个词。响应主体可使用 `response.get_data()` 获取，而 `"Stranger"` 这个词包含在向匿名用户显示的欢迎消息“Hello, Stranger!”中。注意，默认情况下 `get_data()` 得到的响应主体是一个字节数组，传入参数 `as_text=True` 后得到的是一个更易于处理的 Unicode 字符串。

测试客户端还能使用 `post()` 方法发送包含表单数据的 `POST` 请求，不过提交表单时会有一个小麻烦。Flask-WTF 生成的表单中包含一个隐藏字段，其内容是 CSRF 令牌，需要和表单中的数据一起提交。为了复现这个功能，测试必须请求包含表单的页面，然后解析响应返回的 HTML 代码并提取令牌，这样才能把令牌和表单中的数据一起发送。为了避免在测试中处理 CSRF 令牌这一烦琐操作，最好在测试配置中禁用 CSRF 保护功能，如示例 15-3 所示。

示例 15-3　config.py：在测试配置中禁用 CSRF 保护

```
class TestingConfig(Config):
    #...
    WTF_CSRF_ENABLED = False
```

示例 15-4 是一个更为高级的单元测试，模拟了新用户注册账户、登录、使用确认令牌确认账户以及退出的过程。

示例 15-4　tests/test_client.py：使用 Flask 测试客户端模拟新用户注册的整个流程

```
class FlaskClientTestCase(unittest.TestCase):
    # ...
    def test_register_and_login(self):
        # 注册新账户
        response = self.client.post(url_for('auth.register'), data={
            'email': 'john@example.com',
            'username': 'john',
            'password': 'cat',
```

```
        'password2': 'cat'
    })
    self.assertTrue(response.status_code == 302)

    # 使用新注册的账户登录
    response = self.client.post(url_for('auth.login'), data={
        'email': 'john@example.com',
        'password': 'cat'
    }, follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertTrue(re.search('Hello,\s+john!', data))
    self.assertTrue('You have not confirmed your account yet' in data)

    # 发送确认令牌
    user = User.query.filter_by(email='john@example.com').first()
    token = user.generate_confirmation_token()
    response = self.client.get(url_for('auth.confirm', token=token),
                               follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertTrue('You have confirmed your account' in data)

    # 退出
    response = self.client.get(url_for('auth.logout'),
                               follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertTrue('You have been logged out' in data)
```

这个测试先向注册路由提交一个表单。`post()` 方法的 `data` 参数是个字典，包含表单中的各个字段，各字段的名字必须严格匹配定义表单时使用的名字。由于 CSRF 保护已经在测试配置中禁用了，因此无需和表单数据一起发送。

/auth/register 路由有两种响应方式。如果注册数据可用，会返回一个重定向，把用户转到登录页面。注册不可用的情况下，返回的响应会再次渲染注册表单，而且还包含适当的错误消息。为了确认注册成功，测试会检查响应的状态码是否为 302，这个代码表示重定向。

这个测试的第二部分使用刚才注册时使用的电子邮件和密码登录程序。这一工作通过向 /auth/login 路由发起 `POST` 请求完成。这一次，调用 `post()` 方法时指定了参数 `follow_redirects=True`，让测试客户端和浏览器一样，自动向重定向的 URL 发起 `GET` 请求。指定这个参数后，返回的不是 302 状态码，而是请求重定向的 URL 返回的响应。

成功登录后的响应应该是一个页面，显示一个包含用户名的欢迎消息，并提醒用户需要进行账户确认才能获得权限。为此，两个断言语句被用于检查响应是否为这个页面。值得注意的一点是，直接搜索字符串 `'Hello, john!'` 并没有用，因为这个字符串由动态部分和静态部分组成，而且两部分之间有额外的空白。为了避免测试时空白引起的问题，我们使用更为灵活的正则表达式。

下一步我们要确认账户，这里也有一个小障碍。在注册过程中，通过电子邮件将确认 URL

发给用户，而在测试中处理电子邮件不是一件简单的事。上面这个测试使用的解决方法忽略了注册时生成的令牌，直接在 User 实例上调用方法重新生成一个新令牌。在测试环境中，Flask-Mail 会保存邮件正文，所以还有一种可行的解决方法，即通过解析邮件正文来提取令牌。

得到令牌后，测试的第三部分模拟用户点击确认令牌 URL。这一过程通过向确认 URL 发起 GET 请求并附上确认令牌来完成。这个请求的响应是重定向，转到首页，但这里再次指定了参数 follow_redirects=True，所以测试客户端会自动向重定向的页面发起请求。此外，还要检查响应中是否包含欢迎消息和一个向用户说明确认成功的 Flash 消息。

这个测试的最后一步是向退出路由发送 GET 请求，为了证实成功退出，这段测试在响应中搜索一个 Flash 消息。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 git checkout 15b 签出程序的这个版本。

15.2.2　测试Web服务

Flask 测试客户端还可用来测试 REST Web 服务。示例 15-5 是一个单元测试示例，包含了两个测试。

示例 15-5　tests/test_api.py：使用 Flask 测试客户端测试 REST API

```
class APITestCase(unittest.TestCase):
    # ...
    def get_api_headers(self, username, password):
        return {
            'Authorization':
                'Basic ' + b64encode(
                    (username + ':' + password).encode('utf-8')).decode('utf-8'),
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }

    def test_no_auth(self):
        response = self.client.get(url_for('api.get_posts'),
                                   content_type='application/json')
        self.assertTrue(response.status_code == 401)

    def test_posts(self):
        # 添加一个用户
        r = Role.query.filter_by(name='User').first()
        self.assertIsNotNone(r)
        u = User(email='john@example.com', password='cat', confirmed=True,
                 role=r)
```

```
        db.session.add(u)
        db.session.commit()

        # 写一篇文章
        response = self.client.post(
            url_for('api.new_post'),
            headers=self.get_auth_header('john@example.com', 'cat'),
            data=json.dumps({'body': 'body of the *blog* post'}))
        self.assertTrue(response.status_code == 201)
        url = response.headers.get('Location')
        self.assertIsNotNone(url)

        # 获取刚发布的文章
        response = self.client.get(
            url,
            headers=self.get_auth_header('john@example.com', 'cat'))
        self.assertTrue(response.status_code == 200)
        json_response = json.loads(response.data.decode('utf-8'))
        self.assertTrue(json_response['url'] == url)
        self.assertTrue(json_response['body'] == 'body of the *blog* post')
        self.assertTrue(json_response['body_html'] ==
                        '<p>body of the <em>blog</em> post</p>')
```

测试 API 时使用的 `setUp()` 和 `tearDown()` 方法和测试普通程序所用的一样，不过 API 不使用 cookie，所以无需配置相应支持。`get_api_headers()` 是一个辅助方法，返回所有请求都要发送的通用首部，其中包含认证密令和 MIME 类型相关的首部。大多数测试都要发送这些首部。

`test_no_auth()` 是一个简单的测试，确保 Web 服务会拒绝没有提供认证密令的请求，返回 401 错误码。`test_posts()` 测试把一个用户插入数据库，然后使用基于 REST 的 API 创建一篇博客文章，然后再读取这篇文章。所有请求主体中发送的数据都要使用 `json.dumps()` 方法进行编码，因为 Flask 测试客户端不会自动编码 JSON 格式数据。类似地，返回的响应主体也是 JSON 格式，处理之前必须使用 `json.loads()` 方法解码。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 15c` 签出程序的这个版本。

15.3 使用Selenium进行端到端测试

Flask 测试客户端不能完全模拟运行中的程序所在的环境。例如，如果依赖运行在客户端浏览器中的 JavaScript 代码，任何程序都无法正常工作，因为响应发给测试的 JavaScript 代码无法像在真正的 Web 浏览器客户端中那样运行。

如果测试需要完整的环境，除了使用真正的 Web 浏览器连接 Web 服务器中运行的程序外，别无他选。幸运的是，大多数浏览器都支持自动化操作。Selenium（http://www.seleniumhq.org/）是一个 Web 浏览器自动化工具，支持 3 种主要操作系统中的大多数主流 Web 浏览器。

Selenium 的 Python 接口使用 pip 进行安装：

```
(venv) $ pip install selenium
```

使用 Selenium 进行的测试要求程序在 Web 服务器中运行，监听真实的 HTTP 请求。本节使用的方法是，让程序运行在后台线程里的开发服务器中，而测试运行在主线程中。在测试的控制下，Selenium 启动 Web 浏览器并连接程序以执行所需操作。

使用这种方法要解决一个问题，即当所有测试都完成后，要停止 Flask 服务器，而且最好使用一种优雅的方式，以便代码覆盖检测引擎等后台作业能够顺利完成。Werkzeug Web 服务器本身就有停止选项，但由于服务器运行在单独的线程中，关闭服务器的唯一方法是发送一个普通的 HTTP 请求。示例 15-6 实现了关闭服务器的路由。

示例 15-6　app/main/views.py：关闭服务器的路由

```
@main.route('/shutdown')
def server_shutdown():
    if not current_app.testing:
        abort(404)
    shutdown = request.environ.get('werkzeug.server.shutdown')
    if not shutdown:
        abort(500)
    shutdown()
    return 'Shutting down...'
```

只有当程序运行在测试环境中时，这个关闭服务器的路由才可用，在其他配置中调用时将不起作用。在实际过程中，关闭服务器时要调用 Werkzeug 在环境中提供的关闭函数。调用这个函数且请求处理完成后，开发服务器就知道自己需要优雅地退出了。

示例 15-7 是使用 Selenium 运行测试时测试用例所用的代码结构。

示例 15-7　tests/test_selenium.py：使用 Selenium 运行测试的框架

```
from selenium import webdriver

class SeleniumTestCase(unittest.TestCase):
    client = None

    @classmethod
    def setUpClass(cls):
        # 启动 Firefox
        try:
            cls.client = webdriver.Firefox()
        except:
            pass
```

```
        # 如果无法启动浏览器，则跳过这些测试
        if cls.client:
            # 创建程序
            cls.app = create_app('testing')
            cls.app_context = cls.app.app_context()
            cls.app_context.push()

            # 禁止日志，保持输出简洁
            import logging
            logger = logging.getLogger('werkzeug')
            logger.setLevel("ERROR")

            # 创建数据库，并使用一些虚拟数据填充
            db.create_all()
            Role.insert_roles()
            User.generate_fake(10)
            Post.generate_fake(10)

            # 添加管理员
            admin_role = Role.query.filter_by(permissions=0xff).first()
            admin = User(email='john@example.com',
                         username='john', password='cat',
                         role=admin_role, confirmed=True)
            db.session.add(admin)
            db.session.commit()

            # 在一个线程中启动 Flask 服务器
            threading.Thread(target=cls.app.run).start()

    @classmethod
    def tearDownClass(cls):
        if cls.client:
            # 关闭 Flask 服务器和浏览器
            cls.client.get('http://localhost:5000/shutdown')
            cls.client.close()

            # 销毁数据库
            db.drop_all()
            db.session.remove()

            # 删除程序上下文
            cls.app_context.pop()

    def setUp(self):
        if not self.client:
            self.skipTest('Web browser not available')

    def tearDown(self):
        pass
```

setUpClass() 和 tearDownClass() 类方法分别在这个类中的全部测试运行前、后执行。setUpClass() 方法使用 Selenium 提供的 webdriverAPI 启动一个 Firefox 实例，并创建一个程序和数据库，其中写入了一些供测试使用的初始数据。然后调用标准的 app.run() 方法

在一个线程中启动程序。完成所有测试后，程序会收到一个发往 /shutdown 的请求，进而停止后台线程。随后，关闭浏览器，删除测试数据库。

> Selenium 支持 Firefox 之外的很多 Web 浏览器。如果你想使用其他 Web 浏览器，请查阅 Selenium 文档（http://docs.seleniumhq.org/docs/）。

`setUp()` 方法在每个测试运行之前执行，如果 Selenium 无法利用 `startUpClass()` 方法启动 Web 浏览器就跳过测试。示例 15-8 是一个使用 Selenium 进行测试的例子。

示例 15-8 tests/test_selenium.py：Selenium 单元测试示例

```
class SeleniumTestCase(unittest.TestCase):
    # ...

    def test_admin_home_page(self):
        # 进入首页
        self.client.get('http://localhost:5000/')
        self.assertTrue(re.search('Hello,\s+Stranger!',
                                  self.client.page_source))

        # 进入登录页面
        self.client.find_element_by_link_text('Log In').click()
        self.assertTrue('<h1>Login</h1>' in self.client.page_source)

        # 登录
        self.client.find_element_by_name('email').\
            send_keys('john@example.com')
        self.client.find_element_by_name('password').send_keys('cat')
        self.client.find_element_by_name('submit').click()
        self.assertTrue(re.search('Hello,\s+john!', self.client.page_source))

        # 进入用户个人资料页面
        self.client.find_element_by_link_text('Profile').click()
        self.assertTrue('<h1>john</h1>' in self.client.page_source)
```

这个测试使用 `setUpClass()` 方法中创建的管理员账户登录程序，然后打开资料页。注意，这里使用的测试方法和使用 Flask 测试客户端时不一样。使用 Selenium 进行测试时，测试向 Web 浏览器发出指令且从不直接和程序交互。发给浏览器的指令和真实用户使用鼠标或键盘执行的操作几乎一样。

这个测试首先调用 `get()` 方法访问程序的首页。在浏览器中，这个操作就是在地址栏中输入 URL。为了验证这一步操作的结果，测试代码检查页面源码中是否包含“Hello, Stranger!”这个欢迎消息。

为了访问登录页面，测试使用 `find_element_by_link_text()` 方法查找“Log In”链接，然

后在这个链接上调用 `click()` 方法，从而在浏览器中触发一次真正的点击。Selenium 提供了很多 `find_element_by...()` 简便方法，可使用不同的方式搜索元素。

为了登录程序，测试使用 `find_element_by_name()` 方法通过名字找到表单中的电子邮件和密码字段，然后再使用 `send_keys()` 方法在各字段中填入值。表单的提交通过在提交按钮上调用 `click()` 方法完成。此外，还要检查针对用户定制的欢迎消息，以确保登录成功且浏览器显示的是首页。

测试的最后一部分是找到导航条中的“Profile”链接，然后点击。为证实资料页已经加载，测试要在页面源码中搜索内容为用户名的标题。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 15d` 签出程序的这个版本。这次更新包含了一个数据库迁移，所以签出代码后记得要运行 `python manage.py db upgrade`。为保证安装了所有依赖，你还要运行 `pip install -r requirements/dev.txt`。

15.4 值得测试吗

读到这里你可能会问，为了测试而如此折腾 Flask 测试客户端和 Selenium，值得吗？这是一个合理的疑问，不过不容易回答。

不管你是否喜欢，程序肯定要做测试。如果你自己不做测试，用户就要充当不情愿的测试员，用户发现问题后，你就要顶着压力进行修正。检查数据库模型和其他无需在程序上下文中执行的代码很简单，而且有针对性，这类测试一定要做，因为你无需投入过多精力就能保证程序逻辑的核心功能可以正常运行。

我们有时候也需要使用 Flask 测试客户端和 Selenium 进行端到端形式的测试，不过这类测试编写起来比较复杂，只适用于无法进行单独测试的功能。程序代码应该进行合理组织，尽量把业务逻辑写入数据库模型或独立于程序上下文的辅助类中，这样测试起来才更简单。视图函数中的代码应该保持简洁，仅发挥粘合剂的作用，收到请求后调用其他类中对应的操作或者封装程序逻辑的函数。

因此，测试绝对值得。重要的是我们要设计一个高效的测试策略，还要编写能合理利用这一策略的代码。

第 16 章

性能

没人喜欢使用运行缓慢的程序。页面加载时间太长会让用户失去兴趣，所以尽早发现并修正性能问题是一件很重要的工作。在本章，我们要探讨影响性能的两个重要因素。

16.1　记录影响性能的缓慢数据库查询

如果程序性能随着时间推移不断降低，那很有可能是因为数据库查询变慢了，随着数据库规模的增长，这一情况还会变得更糟。优化数据库有时很简单，只需添加更多的索引即可；有时却很复杂，需要在程序和数据库之间加入缓存。大多数数据库查询语言都提供了 `explain` 语句，用来显示数据库执行查询时采取的步骤。从这些步骤中，我们经常能发现数据库或索引设计的不足之处。

不过，在开始优化查询之前，我们必须要知道哪些查询是值得优化的。在一次典型请求中，可能要执行多条数据库查询，所以经常很难分辨哪一条查询较慢。Flask-SQLAlchemy 提供了一个选项，可以记录请求中执行的与数据库查询相关的统计数字。在示例 16-1 中，我们可以看到如何使用这个功能把慢于设定阈值的查询写入日志。

示例 16-1　app/main/views.py：报告缓慢的数据库查询

```
from flask.ext.sqlalchemy import get_debug_queries

@main.after_app_request
def after_request(response):
    for query in get_debug_queries():
        if query.duration >= current_app.config['FLASKY_SLOW_DB_QUERY_TIME']:
            current_app.logger.warning(
                'Slow query: %s\nParameters: %s\nDuration: %fs\nContext: %s\n' %
```

```
                    (query.statement, query.parameters, query.duration,
                        query.context))
    return response
```

这个功能使用 after_app_request 处理程序实现，它和 before_app_request 处理程序的工作方式类似，只不过在视图函数处理完请求之后执行。Flask 把响应对象传给 after_app_request 处理程序，以防需要修改响应。

在本例中，after_app_request 处理程序没有修改响应，只是获取 Flask-SQLAlchemy 记录的查询时间并把缓慢的查询写入日志。

get_debug_queries() 函数返回一个列表，其元素是请求中执行的查询。Flask-SQLAlchemy 记录的查询信息如表 16-1 所示。

表16-1　Flask-SQLAlchemy记录的查询信息

名　　称	说　　明
statement	SQL 语句
parameters	SQL 语句使用的参数
start_time	执行查询时的时间
end_time	返回查询结果时的时间
duration	查询持续的时间，单位为秒
context	表示查询在源码中所处位置的字符串

after_app_request 处理程序遍历 get_debug_queries() 函数获取的列表，把持续时间比设定阈值长的查询写入日志。写入的日志被设为“警告”等级。如果换成“错误”等级，发现缓慢的查询时还会发送电子邮件。

默认情况下，get_debug_queries() 函数只在调试模式中可用。但是数据库性能问题很少发生在开发阶段，因为开发过程中使用的数据库较小。因此，在生产环境中使用该选项才能发挥作用。若想在生产环境中分析数据库性能，我们必须修改配置，如示例 16-2 所示。

示例 16-2　config.py：启用缓慢查询记录功能的配置

```
class Config:
    # ...
    SQLALCHEMY_RECORD_QUERIES = True
    FLASKY_DB_QUERY_TIMEOUT = 0.5
    # ...
```

SQLALCHEMY_RECORD_QUERIES 告诉 Flask-SQLAlchemy 启用记录查询统计数字的功能。缓慢查询的阈值设为 0.5 秒。这两个配置变量都在 Config 基类中设置，因此在所有环境中都可使用。

每当发现缓慢查询，Flask 程序的日志记录器就会写入一条记录。若想保存这些日志记录，必须配置日志记录器。日志记录器的配置根据程序所在主机使用的平台而有所不同，第 17 章会举一些例子。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 16a` 签出程序的这个版本。

16.2 分析源码

性能问题的另一个可能诱因是高 CPU 消耗，由执行大量运算的函数导致。源码分析器能找出程序中执行最慢的部分。分析器监视运行中的程序，记录调用的函数以及运行各函数所消耗的时间，然后生成一份详细的报告，指出运行最慢的函数。

> 分析一般在开发环境中进行。源码分析器会让程序运行得更慢，因为分析器要监视并记录程序中发生的一切。因此我们不建议在生产环境中进行分析，除非使用专为生产环境设计的轻量级分析器。

Flask 使用的开发 Web 服务器由 Werkzeug 提供，可根据需要为每条请求启用 Python 分析器。示例 16-3 向程序中添加了一个新命令，用来启动分析器。

示例 16-3　manage.py：在请求分析器的监视下运行程序

```
@manager.command
def profile(length=25, profile_dir=None):
    """Start the application under the code profiler."""
    from werkzeug.contrib.profiler import ProfilerMiddleware
    app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions=[length],
                                      profile_dir=profile_dir)
    app.run()
```

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 16b` 签出程序的这个版本。

使用 `python manage.py profile` 启动程序后，终端会显示每条请求的分析数据，其中包含运行最慢的 25 个函数。`--length` 选项可以修改报告中显示的函数数量。如果指定了

`--profile-dir` 选项，每条请求的分析数据就会保存到指定目录下的一个文件中。分析器数据文件可用来生成更详细的报告，例如调用图。Python 分析器的详细信息请参阅官方文档（https://docs.python.org/2/library/profile.html）。

现在我们完成了部署前的准备工作。在下一章，你会了解部署程序的大致过程。

第 17 章

部署

Flask 自带的开发 Web 服务器不够强健、安全和高效，无法在生产环境中使用。在本章，我们要介绍几种不同的部署方式。

17.1　部署流程

不管使用哪种托管方案，程序安装到生产服务器上之后，都要执行一系列的任务。最好的例子就是创建或更新数据库表。

如果每次安装或升级程序都手动执行任务，那么容易出错也浪费时间，所以我们可以在 manage.py 中添加一个命令，自动执行所需操作。

示例 17-1 实现了一个适用于 Flasky 的 `deploy` 命令。

示例 17-1　manage.py：部署命令

```
@manager.command
def deploy():
    """Run deployment tasks."""
    from flask.ext.migrate import upgrade
    from app.models import Role, User

    # 把数据库迁移到最新修订版本
    upgrade()

    # 创建用户角色
    Role.insert_roles()
```

```
    # 让所有用户都关注此用户
    User.add_self_follows()
```

这个命令调用的函数之前都已经定义好了，现在只是将它们集中调用。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 17a` 签出程序的这个版本。

定义这些函数时考虑到了多次执行的情况，所以即使多次执行也不会产生问题。因此每次安装或升级程序时只需运行 `deploy` 命令就能完成所有操作。

17.2　把生产环境中的错误写入日志

如果调试模式中运行的程序发生错误，那么会出现 Werkzeug 中的交互式调试器。网页中显示错误的*栈跟踪*，而且可以查看源码，甚至还能使用 Flask 的网页版交互调试器在每个栈帧的上下文中执行表达式。

调试器是开发过程中进行问题调试的优秀工具，但其显然不能在生产环境中使用。生产环境中发生的错误会被静默掉，取而代之的是向用户显示一个 500 错误页面。不过幸好错误的栈跟踪不会完全丢失，因为 Flask 会将其写入*日志文件*。

在程序启动过程中，Flask 会创建一个 Python 提供的 `logging.Logger` 类实例，并将其附属到程序实例上，得到 `app.logger`。在调试模式中，日志记录器会把记录写入终端；但在生产模式中，默认情况下没有配置日志的处理程序，所以如果不添加处理程序，就不会保存日志。示例 17-2 中的改动配置了一个日志处理程序，把生产模式中出现的错误通过电子邮件发送给 `FLASKY_ADMIN` 中设置的管理员。

示例 17-2　config.py：程序出错时发送电子邮件

```
class ProductionConfig(Config):
    # ...
    @classmethod
    def init_app(cls, app):
        Config.init_app(app)

        # 把错误通过电子邮件发送给管理员
        import logging
        from logging.handlers import SMTPHandler
        credentials = None
        secure = None
        if getattr(cls, 'MAIL_USERNAME', None) is not None:
            credentials = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD)
            if getattr(cls, 'MAIL_USE_TLS', None):
```

```
        secure = ()
    mail_handler = SMTPHandler(
        mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT),
        fromaddr=cls.FLASKY_MAIL_SENDER,
        toaddrs=[cls.FLASKY_ADMIN],
        subject=cls.FLASKY_MAIL_SUBJECT_PREFIX + ' Application Error',
        credentials=credentials,
        secure=secure)
    mail_handler.setLevel(logging.ERROR)
    app.logger.addHandler(mail_handler)
```

回顾一下，所有配置实例都有一个 `init_app()` 静态方法，在 `create_app()` 方法中调用。在 `ProductionConfig` 类的 `init_app()` 方法的实现中，配置程序的日志记录器把错误写入电子邮件日志记录器。

电子邮件日志记录器的日志等级被设为 `logging.ERROR`，所以只有发生严重错误时才会发送电子邮件。通过添加适当的日志处理程序，可以把较轻缓等级的日志消息写入文件、系统日志或其他的支持方法。这些日志消息的处理方法很大程度上依赖于程序使用的托管平台。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 17b` 签出程序的这个版本。

17.3 云部署

程序托管的最新潮流是托管到云端。云技术以前称为平台即服务（Platform as a Service，PaaS），它让程序开发者从安装和维护运行程序的软硬件平台的日常工作中解放出来。在 PaaS 模型中，服务提供商完全接管了运行程序的平台。程序开发者使用服务商提供的工具和库把程序集成到平台上，然后将其上传到提供商维护的服务器中，部署的过程往往只需几秒钟。大多数 PaaS 提供商都可以通过按需添加或删除服务器以实现程序的动态扩展，从而满足不同请求量的需求。

云部署有较高的灵活性，而且使用起来相对容易。当然，这些优势都是花钱买来的。Heroku 是最流行的 PaaS 提供商之一，对 Python 支持良好。下一节，我们将详细说明如何把程序部署到 Heroku 中。

17.4 Heroku平台

Heroku 是最早出现的 PaaS 提供商之一，从 2007 年就开始运营。Heroku 平台的灵活性极高且支持多种编程语言。若想把程序部署到 Heroku 上，开发者要使用 Git 把程序推送到

Heroku 的 Git 服务器上。在服务器上，`git push` 命令会自动触发安装、配置和部署程序。

Heroku 使用名为 Dyno 的计算单元衡量用量，并以此为依据收取服务费用。最常用的 Dyno 类型是 Web Dyno，表示一个 Web 服务器实例。程序可以通过使用更多的 Web Dyno 以增强其请求处理能力。另一种 Dyno 类型是 Worker Dyno，用来执行后台作业或其他辅助任务。

Heroku 提供了大量的插件和扩展，可用于数据库、电子邮件支持和其他很多服务。下面各节将展开说明把 Flasky 部署到 Heroku 上的细节步骤。

17.4.1　准备程序

若想使用 Heroku，程序必须托管在 Git 仓库中。如果你的程序托管在像 GitHub 或 BitBucket 这样的远程 Git 服务器上，那么克隆程序后会创建一个本地 Git 仓库，可无缝用于 Heroku。如果你的程序没有托管在 Git 仓库中，那么必须在开发电脑上创建一个仓库。

> 如果你计划把程序托管在 Heroku 上，最好从开发伊始就使用 Git。GitHub 的帮助指南（http://help.github.com/）中有针对 3 种主流操作系统的安装及设置说明。

1. 注册Heroku账户

在使用 Heroku 提供的服务之前，你必须要注册一个账户（http://heroku.com/）。注册后，你可以选择免费的最低等级的服务托管程序，因此，Heroku 非常适合做实验。

2. 安装Heroku Toolbelt

最方便的 Heroku 程序管理方法是使用 Heroku Toolbelt（https://toolbelt.heroku.com/）命令行工具。Toolbelt 由两个 Heroku 程序组成。

- `heroku`：Heroku 客户端，用来创建和管理程序。
- `foreman`：一种工具，测试时可用于在自己的电脑上模拟 Heroku 环境。

注意，如果你之前没有安装 Git 客户端，那么 Toolbelt 安装程序会为你安装 Git。

在 Heroku 客户端连接服务器之前，你需要提供 Heroku 账户密令。`heroku login` 命令可以完成这一操作：

```
$ heroku login
Enter your Heroku credentials.
Email: <your-email-address>
Password (typing will be hidden): <your-password>
Uploading ssh public key .../id_rsa.pub
```

> 把你的 SSH 公钥上传到 Heroku 这一点很重要，上传后才能使用 `git push` 命令。正常情况下，`login` 命令会自动创建并上传 SSH 公钥。但你也可以使用 `heroku keys:add` 命令单独上传公钥或者上传额外所需的公钥。

3. 创建程序

接下来，我们要使用 Heroku 客户端创建一个程序。为此，我们首先要确保程序已纳入 Git 源码控制系统，然后在顶级目录中运行如下命令：

```
$ heroku create <appname>
Creating <appname>... done, stack is cedar
http://<appname>.herokuapp.com/ | git@heroku.com:<appname>.git
Git remote heroku added
```

Heroku 中的程序名必须是唯一的，所以你要找一个没被其他程序使用的名字。如 `create` 命令的输出所示，部署后程序可通过 http://<*appname*>.herokuapp.com 访问。你可以给程序设置自定义域名。

在程序创建过程中，Heroku 还给你分配了一个 Git 服务器，地址为 git@heroku.com:<*appname*>.git。`create` 命令调用 `git remote` 命令把这个地址添加为本地 Git 仓库的远程服务器，名为 `heroku`。

4. 配置数据库

Heroku 以扩展形式支持 Postgres 数据库。少于 1 万条记录的小型数据库无需付费即可添加到程序中：

```
$ heroku addons:add heroku-postgresql:dev
Adding heroku-postgresql:dev on <appname>... done, v3 (free)
Attached as HEROKU_POSTGRESQL_BROWN_URL
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pgbackups:restore.
Use`heroku addons:docs heroku-postgresql:dev`to view documentation.
```

环境变量 `HEROKU_POSTGRESQL_BROWN_URL` 中保存了数据库的 URL。注意，运行这个命令后，你得到的颜色可能不是棕色。Heroku 中的每个程序都支持多个数据库，而每个数据库 URL 中的颜色都不一样。数据库的地位可以提升，把 URL 保存到环境变量 `DATABASE_URL` 中。下述命令把前面创建的棕色数据库提升为主数据库：

```
$ heroku pg:promote HEROKU_POSTGRESQL_BROWN_URL
Promoting HEROKU_POSTGRESQL_BROWN_URL to DATABASE_URL... done
```

`DATABASE_URL` 环境变量的格式正是 SQLAlchemy 所需的。回想一下 config.py 脚本的内容，如果设定了 `DATABASE_URL`，就使用其中保存的值，所以现在程序可以自动连接到 Postgres 数据库。

5. 配置日志

之前我们实现了通过电子邮件发送重大错误消息的功能，除此之外，配置其他轻缓等级的消息也尤为重要。其中一个很好的例子是第 16 章添加的数据库缓慢查询警告消息。

在 Heroku 中，日志必须写入 `stdout` 或 `stderr`。Heroku 会捕获输出的日志，可以在 Heroku 客户端中使用 `heroku logs` 命令查看。

日志的配置可添加到 `ProductionConfig` 类的 `init_app()` 静态方法中，但由于这种日志处理方式是 Heroku 专用的，因此可专门为这个平台新建一个配置类，把 `ProductionConfig` 作为不同类型生产平台的基类。`HerokuConfig` 类如示例 17-3 所示。

示例 17-3　config.py：Heroku 的配置

```
class HerokuConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # 输出到 stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler()
        file_handler.setLevel(logging.WARNING)
        app.logger.addHandler(file_handler)
```

通过 Heroku 执行程序时，程序需要知道要使用的就是这个配置。manage.py 脚本创建的程序实例通过环境变量 `FLASK_CONFIG` 决定使用哪个配置，所以我们要在 Heroku 的环境中设定这个变量。环境变量使用 Heroku 客户端中的 `config:set` 命令设定：

```
$ heroku config:set FLASK_CONFIG=heroku
Setting config vars and restarting <appname>... done, v4
FLASK_CONFIG: heroku
```

6. 配置电子邮件

Heroku 没有提供 SMTP 服务器，所以我们要配置一个外部服务器。很多第三方扩展能把适用于生产环境的邮件发送服务集成到 Heroku 中，但对于测试和评估而言，使用继承自 `Config` 基类的 Gmail 配置已经足够了。

由于直接把安全密令写入脚本存在安全隐患，所以我们把访问 Gmail SMTP 服务器的用户名和密码保存在环境变量中：

```
$ heroku config:set MAIL_USERNAME=<your-gmail-username>
$ heroku config:set MAIL_PASSWORD=<your-gmail-password>
```

7. 运行生产Web服务器

Heroku 没有为托管程序提供 Web 服务器，相反，它希望程序启动自己的服务器并监听环境变量 `PORT` 中设定的端口。

Flask 自带的开发 Web 服务器表现很差，因为它不是为生产环境设计的服务器。有两个可以在生产环境中使用、性能良好且支持 Flask 程序的服务器，分别是 Gunicorn（http://gunicorn.org/）和 uWSGI（http://uwsgi-docs.readthedocs.org/en/latest/）。

若想在本地测试 Heroku 配置，我们最好在虚拟环境中安装 Web 服务器。例如，可通过如下命令安装 Gunicorn：

```
(venv) $ pip install gunicorn
```

若要使用 Gunicorn 运行程序，可执行下面的命令：

```
(venv) $ gunicorn manage:app
2013-12-03 09:52:10 [14363] [INFO] Starting gunicorn 18.0
2013-12-03 09:52:10 [14363] [INFO] Listening at: http://127.0.0.1:8000 (14363)
2013-12-03 09:52:10 [14363] [INFO] Using worker: sync
2013-12-03 09:52:10 [14368] [INFO] Booting worker with pid: 14368
```

`manage:app` 参数冒号左边的部分表示定义程序的包或者模块，冒号右边的部分表示包中程序实例的名字。注意，Gunicor 默认使用端口 8000，而 Flask 默认使用 5000。

8. 添加依赖需求文件

Heroku 从程序顶级文件夹下的 requirements.txt 文件中加载包依赖。这个文件中的所有依赖都会在部署过程中导入 Heroku 创建的虚拟环境。

Heroku 的需求文件必须包含程序在生产环境中使用的所有通用依赖，以及支持 Postgres 数据库的 psycopg2 包和 Gunicorn Web 服务器。

示例 17-4 是一个需求文件的例子。

示例 17-4　requirements.txt：Heroku 需求文件

```
-r requirements/prod.txt
gunicorn==18.0
psycopg2==2.5.1
```

9. 添加Procfile文件

Heroku 需要知道使用哪个命令启动程序。这个命令在一个名为 Procfile 的特殊文件中指定。这个文件必须放在程序的顶级文件夹中。

示例 17-5 展示了这个文件的内容。

示例 17-5　Procfile：Heroku Procfile 文件

```
web: gunicorn manage:app
```

Procfile 文件内容的格式很简单：在每一行中指定一个任务名，后跟一个冒号，然后是运行这个任务的命令。名为 `web` 的任务比较特殊任务，Heroku 使用这个任务启动 Web 服务器。

Heroku 会为这个任务提供一个 `PORT` 环境变量，用于设定程序监听请求的端口。如果设定了 `PORT` 变量，Gunicorn 默认就会使用其中保存的值，因此无需将其包含在启动命令中。

> 程序可在 Procfile 中使用 `web` 之外的名字声明其他任务，例如程序所需的其他服务。部署程序后，Heroku 会运行 Procfile 中列出的所有任务。

17.4.2 使用Foreman进行测试

Heroku Toolbelt 中还包含另一个名为 Foreman 的工具，它用于在本地通过 Procfile 运行程序以进行测试。Heroku 客户端设定的像 `FLASK_CONFIG` 这样的环境变量只在 Heroku 服务器上可用，因此要在本地设定，这样 Foreman 使用的测试环境才和生产环境类似。Foreman 会在程序顶级目录中搜寻一个名为 .env 的文件，加载其中的环境变量。例如 .env 文件中可包含以下变量：

```
FLASK_CONFIG=heroku
MAIL_USERNAME=<your-username>
MAIL_PASSWORD=<your-password>
```

> 由于 .env 文件中包含密码和其他敏感的账户信息，所以决不能将其添加到 Git 仓库中。

Foreman 有多个命令，其中两个主要命令是 `foreman run` 和 `foreman start`。`run` 命令用于在程序的环境中运行任意命令，特别适合运行创建程序数据库的 `deploy` 命令：

```
(venv) $ foreman run python manage.py deploy
```

`start` 命令读取 Procfile 的内容，执行其中的所有任务：

```
(venv) $ foreman start
22:55:08 web.1  | started with pid 4246
22:55:08 web.1  | 2013-12-03 22:55:08 [4249] [INFO] Starting gunicorn 18.0
22:55:08 web.1  | 2013-12-03 22:55:08 [4249] [INFO] Listening at: http://...
22:55:08 web.1  | 2013-12-03 22:55:08 [4249] [INFO] Using worker: sync
22:55:08 web.1  | 2013-12-03 22:55:08 [4254] [INFO] Booting worker with pid: 4254
```

Foreman 把所有启动任务的日志输出整合在一起并转储至终端，其中每一行的前面都加入了时间戳和任务名。

使用 `-c` 选项还能模拟多个 Dyno。例如，下述命令启动了 3 个 Web 工作线程（Web

worker），各职程分别监听不同的端口：

```
(venv) $ foreman start -c web=3
```

17.4.3　使用Flask-SSLify启用安全HTTP

用户登录程序时要在 Web 表单中提交用户名和密码，这些数据在传输过程中可被第三方截取，就像前文已多次提及的。为了避免他人使用这种方式偷取用户密令，我们必须使用安全 HTTP，使用公钥加密法加密客户端和服务器之间传输的数据。

Heroku 上的程序在 herokuapp.com 域中可使用 http:// 和 https:// 访问，无需任何配置即可直接使用 Heroku 的 SSL 证书。唯一需要做的是让程序拦截发往 http:// 的请求，重定向到 https://，这一操作可使用 Flask-SSLify 扩展完成。

我们要将 Flask-SSLify 扩展添加到 requirements.txt 文件中。示例 17-6 中的代码用于激活这个扩展。

示例 17-6　app/__init__.py：把所有请求重定向到安全 HTTP

```
def create_app(config_name):
    # ...
    if not app.debug and not app.testing and not app.config['SSL_DISABLE']:
        from flask.ext.sslify import SSLify
        sslify = SSLify(app)
    # ...
```

对 SSL 的支持只需在生产模式中启用，而且所在平台必须支持。为了便于打开和关闭 SSL，添加了一个名为 `SSL_DISABLE` 的新配置变量。`Config` 基类将其设为 `True`，即默认情况下不使用 SSL，并且 `HerokuConfig` 类覆盖了这个值。这个变量的配置方式如示例 17-7 所示。

示例 17-7　config.py：配置是否使用 SSL

```
class Config:
    # ...
    SSL_DISABLE = True

class HerokuConfig(ProductionConfig):
    # ...
    SSL_DISABLE = bool(os.environ.get('SSL_DISABLE'))
```

在 `HerokuConfig` 类中，`SSL_DISABLE` 的值从同名环境变量中读取。如果这个环境变量的值不是空字符串，那么将其转换成布尔值后会得到 `True`，即禁用 SSL。如果没有设定这个环境变量或者其值为空字符串，转换成布尔值后会得到 `False`。为了避免使用 Foreman 时启用 SSL，必须在 .env 文件中加入 `SSL_DISABLE=1`。

做了以上改动后，用户会被强制使用 SSL。但还有一个细节需要处理才能完善这一功能。使用 Heroku 时，客户端不直接连接托管的程序，而是连接一个*反向代理服务器*，然后再

把请求重定向到程序上。在这种连接方式中，只有代理服务器运行在 SSL 模式中。程序从代理服务器接收到的请求都没有使用 SSL，因为在 Heroku 网络内部无需使用高安全性的请求。程序生成绝对 URL 时，要和请求使用的安全连接一致，这时就产生问题了，因为使用反向代理服务器时，`request.is_secure` 的值一直是 `False`。

这个问题会在生成头像的 URL 时发生。回想一下第 10 章的内容，`User` 模型中的 `gravatar()` 方法在生成 Gravatar URL 时检查了 `request.is_secure`，根据其值的不同分别生成安全或不安全的 URL。如果通过 SSL 请求页面，生成的却是不安全的头像 URL，某些浏览器会向用户显示安全警告，所以同一页面中的所有内容都要使用安全性相同的 URL。

代理服务器通过自定义的 HTTP 首部把客户端发起的原始请求信息传给重定向后的 Web 服务器，所以查看这些首部就有可能知道用户和程序通信时是否使用了 SSL。Werkzeug 提供了一个 WSGI 中间件，可用来检查代理服务器发出的自定义首部并对请求对象进行相应更新。例如，修改后的 `request.is_secure` 表示客户端发给反向代理服务器的请求安全性，而不是代理服务器发给程序的请求安全性。示例 17-8 展示了如何把 `ProxyFix` 中间件添加到程序中。

示例 17-8 config.py：支持代理服务器

```
class HerokuConfig(ProductionConfig):
    # ...
    @classmethod
    def init_app(cls, app):
        # ...

        # 处理代理服务器首部
        from werkzeug.contrib.fixers import ProxyFix
        app.wsgi_app = ProxyFix(app.wsgi_app)
```

`ProxyFix` 中间件添加在 Heroku 配置的初始化方法中。添加 `ProxyFix` 等 WSGI 中间件的方法是包装 WSGI 程序。收到请求时，中间件有机会审查环境，在处理请求之前做些修改。不仅 Heroku 需要使用 `ProxyFix` 中间件，任何使用反向代理的部署环境都需要。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 17c` 签出程序的这个版本。为保证安装了所有依赖，你还要运行 `pip install -r requirements.txt`。

17.4.4 执行 `git push` 命令部署

部署过程的最后一步是把程序上传到 Heroku 服务器。在此之前，你要确保所有改动都已经提交到本地 Git 仓库，然后执行 `git push heroku master` 把程序上传到远程仓库 `heroku`：

```
$ git push heroku master
Counting objects: 645, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (315/315), done.
Writing objects: 100% (645/645), 95.52 KiB, done.
Total 645 (delta 369), reused 457 (delta 288)

.---> Python app detected
.----> No runtime.txt provided; assuming python-2.7.4.
.----> Preparing Python runtime (python-2.7.4)
...
-----> Compiled slug size: 32.8MB
-----> Launching... done, v8
       http://<appname>.herokuapp.com deployed to Heroku

To git@heroku.com:<appname>.git
 * [new branch]      master -> master
```

现在，程序已经部署好正在运行了，但还不能正常使用，因为还没执行 `deploy` 命令。Heroku 客户端可按照下面的方式执行这个命令：

```
$ heroku run python manage.py deploy
Running `python manage.py predeploy` attached to terminal... up, run.8449
INFO  [alembic.migration] Context impl PostgresqlImpl.
INFO  [alembic.migration] Will assume transactional DDL.
...
```

创建并配置好数据库表之后就可以重启程序了，直接使用下述命令即可：

```
$ heroku restart
Restarting dynos... done
```

至此，程序就完全部署好了，可通过 https://<*appname*>.hero-kuapp.com 访问。

17.4.5　查看日志

程序生成的日志输出会被 Heroku 捕获，若想查看日志内容，可使用 `logs` 命令：

```
$ heroku logs
```

在测试过程中，还可以使用下述命令方便地跟踪日志文件的内容：

```
$ heroku logs -t
```

17.4.6　部署一次升级

升级 Heroku 程序时要重复上述步骤。所有改动都提交到 Git 仓库之后，可执行下述命令进行升级：

```
$ heroku maintenance:on
$ git push heroku master
$ heroku run python manage.py deploy
$ heroku restart
$ heroku maintenance:off
```

Heroku 客户端提供的 `maintenance` 命令会在升级过程中下线程序，并向用户显示一个静态页面，告知网站很快就能恢复。

17.5　传统的托管

如果你选择使用传统托管，那么要购买或租用服务器（物理服务器或虚拟服务器），然后自己动手在服务器上设置所有需要的组件。传统托管一般比托管在云中要便宜，但显然要付出更多的劳动。下面各节将简要说明其中涉及的工作。

17.5.1　架设服务器

在能够托管程序之前，服务器必须完成多项管理任务。

- 安装数据库服务器，例如 MySQL 或 Postgres。也可使用 SQLite 数据库，但由于其自身的种种限制，不建议用于生产服务器。
- 安装邮件传输代理（Mail Transport Agent，MTA），例如 Sendmail，用于向用户发送邮件。
- 安装适用于生产环境的 Web 服务器，例如 Gunicorn 或 uWSGI。
- 为了启用安全 HTTP，购买、安装并配置 SSL 证书。
- （可选，但强烈推荐）安装前端反向代理服务器，例如 nginx 或 Apache。反向代理服务器能直接服务于静态文件，而把其他请求转发给程序使用的 Web 服务器。Web 服务器监听 localhost 中的一个私有端口。
- 强化服务器。这一过程包含多项任务，目标在于降低服务器被攻击的可能性，例如安装防火墙以及删除不用的软件和服务等。

17.5.2　导入环境变量

和 Heroku 中的程序一样，运行在独立服务器上的程序也要依赖某些设置，例如数据库 URL、电子邮件服务器密令以及配置名。这些设置保存在环境变量中，启动服务器之前必须导入。

由于没有 Heroku 客户端和 Foreman 来导入变量，这个任务需要在启动过程中由程序本身完成。示例 17-9 中这段简短的代码能加载并解析 Foreman 使用的 .env 文件。在创建程序实例代码之前，可以将这段代码添加到启动脚本 manage.py 中。

示例 17-9　manage.py：从 .env 文件中导入环境变量

```
if os.path.exists('.env'):
```

```
        print('Importing environment from .env...')
        for line in open('.env'):
            var = line.strip().split('=')
            if len(var) == 2:
                os.environ[var[0]] = var[1]
```

.env 文件中至少要包含 FLASK_CONFIG 变量，用以选择要使用的配置。

17.5.3 配置日志

在基于 Unix 的服务器中，日志可发送给守护进程 syslog。我们可专门为 Unix 创建一个新配置，继承自 ProductionConfig，如示例 17-10 所示。

示例 17-10 config.py：Unix 配置示例

```
class UnixConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # 写入系统日志
        import logging
        from logging.handlers import SysLogHandler
        syslog_handler = SysLogHandler()
        syslog_handler.setLevel(logging.WARNING)
        app.logger.addHandler(syslog_handler)
```

这样配置之后，程序的日志会写入 /var/log/messages。如果需要，我们还可以配置系统日志服务，从而把日志写入别的文件或者发送到其他设备中。

> 如果你从 GitHub 上克隆了这个程序的 Git 仓库，那么可以执行 `git checkout 17d` 签出程序的这个版本。

第 18 章

其他资源

恭喜，你快读完本书了。希望本书涵盖的话题能为你打下坚实的基础，让你开始使用 Flask 开发程序。书中的示例代码是开源的，基于一个宽松的许可协议发布，所以你可以在项目中尽情使用其中代码，即便是用于商业项目。在这最后的简短一章中，我列出了一些建议和资源，希望能为你继续使用 Flask 提供一些帮助。

18.1 使用集成开发环境

在集成开发环境（Integrated Development Environment，IDE）中开发 Flask 程序非常方便，因为代码补全和交互式调试器等功能可以显著提升编程的速度。以下是几个适合进行 Flask 开发的 IDE。

- PyCharm（http://www.jetbrains.com/pycharm/）：JetBrains 开发的商用 IDE，有社区版（免费）和专业版（付费），两个版本都兼容 Flask 程序，可在 Linux、Mac OS X 和 Windows 中使用。
- PyDev（http://pydev.org/）：这是基于 Eclipse 的开源 IDE，可在 Linux、Mac OS X 和 Windows 中使用。
- Python Tools for Visual Studio（http://pytools.codeplex.com/）：这是免费 IDE，作为微软 Visual Studio 的一个扩展，只能在微软 Windows 中使用。

配置 Flask 程序在调试器中启动时，记得为 `runserver` 命令加入 `--passthrough-errors --no-reload` 选项。第一个选项禁用 Flask 对错误的缓存，这样处理请求过程中抛出的异常才会传到调试器中。第二个选项禁用重载模块，而这个模块会搅乱某些调试器。

18.2 查找Flask扩展

本书中的示例程序用到了很多扩展和包，不过还有很多有用的扩展没有介绍。下面列出了其他一些值得研究的包。

- Flask-Babel（https://pythonhosted.org/Flask-Babel/）：提供国际化和本地化支持。
- FLask-RESTful（http://flask-restful.readthedocs.org/en/latest/）：开发 REST API 的工具。
- Celery（http://docs.celeryproject.org/en/latest/）：处理后台作业的任务队列。
- Frozen-Flask（https://pythonhosted.org/Frozen-Flask/）：把 Flask 程序转换成静态网站。
- Flask-DebugToolbar（https://github.com/mgood/flask-debugtoolbar）：在浏览器中使用的调试工具。
- Flask-Assets（https://github.com/miracle2k/flask-assets）：用于合并、压缩、编译 CSS 和 JavaScript 静态资源文件。
- Flask-OAuth（http://pythonhosted.org/Flask-OAuth/）：使用 OAuth 服务进行认证。
- Flask-OpenID（http://pythonhosted.org/Flask-OpenID/）：使用 OpenID 服务进行认证。
- Flask-WhooshAlchemy（https://pythonhosted.org/Flask-WhooshAlchemy/）：使用 Whoosh（http://pythonhosted.org/Whoosh/）实现 Flask-SQLAlchemy 模型的全文搜索。
- Flask-KVsession（http://flask-kvsession.readthedocs.org/en/latest/）：使用服务器端存储实现的另一种用户会话。

如果项目中的某些功能无法使用本书介绍的扩展和包实现，那么你首先可以到 Flask 官方扩展网站（http://flask.pocoo.org/extensions/）查找其他扩展。其他可以搜寻扩展的地方有：Python Package Index（http://pypi.python.org/）、GitHub（http://github.com/） 和 BitBucket（http://bitbucket.org/）。

18.3 参与Flask开发

如果没有社区开发者的贡献，Flask 不会如此优秀。现在你已经成为社区的一份子，也从众多志愿者的劳动中受益，所以你应该考虑通过某种方式来回馈社区。如果你不知从何入手，可考虑下面这些建议：

- 审阅 Flask 或者你最喜欢的相关项目文档，提交修正或改进；
- 把文档翻译成其他语言；
- 在问答网站上回答问题，例如 Stack Overflow（http://stackoverflow.com/）；
- 在用户组的聚会或者会议上和同行讨论你的工作；
- 对于你使用的包中的错误，贡献修正和改进建议；
- 开发新 Flask 扩展，开源发布；
- 开源自己的程序。

希望你能使用上述或者其他有意义的方式为社区做贡献。如果你这么做了，那我由衷地感谢你！

关于封面图

本书封面上的动物是比利牛斯獒犬（家犬的一种）。这种大型西班牙犬的祖先是一种名为马鲁索斯犬的家畜守卫犬，这种犬最早由希腊人和罗马人饲养，现已灭绝。不过，马鲁索斯犬在现今多种常见犬类的繁育过程中都扮演了重要角色，例如罗威那犬、大丹犬、纽芬兰犬和卡斯罗犬。直到 1977 年，比利牛斯獒犬才被确认为纯种犬。美国比利牛斯獒犬俱乐部致力于把这种犬作为宠物在美国推广。

西班牙内战结束后，原产地的比利牛斯獒犬数量急剧下降。这一犬种能幸存下来完全有赖于分散在全国各地的专职饲养员。比利牛斯獒犬的现代基因库源于这一战后种群，所以它们很容易得遗传病，例如髋关节发育不良。现在，负责任的主人都会在饲养前对狗做疾病检查和 X 光照射以排除髋关节异常。

成年雄性比利牛斯獒犬完全长成后可重达 200 英磅，所以饲养这种狗要保证充足的训练和遛狗时间。比利牛斯獒犬虽然体型很大，而且曾作为抵挡熊和狼的猎犬，但其性情温顺，是一种优秀的家犬。人类可以放心地让这种狗照看儿童和守护庭院，而且可以将其和其他狗一起驯养。比利牛斯獒犬有一定的社交能力和较强的领导力，在家庭环境的熏陶之下，它们已经成为一种优秀的守护犬和伙伴。

本书的封面图片出自 Wood 的 *Animate Creation* 一书。

O'REILLY®

Flask Web开发：基于Python的Web应用开发实战

作为Python Web开发的微框架，Flask独树一帜。它不会强迫开发者遵循预置的开发规范，为开发者提供了自由度和创意空间。

本书作者拥有25年软件开发经验，而本书则采用讲解与实例相结合的方式，不仅介绍了Flask安装、使用等基础知识，而且还带领读者一步一步地开发了社交博客Flasky。即使从未接触Flask，你也能轻松学会构建完整的Web应用。通读本书，你能熟悉Flask的核心功能，并掌握数据库迁移、Web服务通信等高级Web技术。

本书不仅适合初级Web开发人员学习阅读，更是Python程序员用来学习高级Web开发技术的优秀参考书。

- 学习Flask应用的基本结构，编写示例应用；
- 使用必备的组件，包括模板、数据库、Web表单和电子邮件支持；
- 使用包和模块构建可伸缩的大型应用；
- 实现用户认证、角色和个人资料；
- 在博客网站中重用模板、分页显示列表以及使用富文本；
- 使用基于Flask的REST式API，在智能手机、平板电脑和其他第三方客户端上实现可用功能；
- 学习运行单元测试以及提升性能；
- 将Web应用部署到生产服务器。

“好久没有看到这么棒的技术书了！它从安装与环境设置讲起，目标则是搭建服务器端Web应用。本书直接了当地给出了读者必知必会的知识，为初学者提供了进一步探索的起点，也让中高级读者能够掌握最佳实践。”

“我不是新手，做过Flask应用开发，我以为自己完全了解相关基础知识。但实际阅读中，我却折了很多页，时不时会翻阅相关的知识点。其中的技巧和提示总能让我茅塞顿开……真是有幸读了这本书！”

“本书的组织结构非常合理。读完本书，我也亲身参与构建了Web应用，真正拾级而上掌握了强大的Flask开发。”

Miguel Grinberg 拥有25年开发经验的高级软件工程师，目前为广播公司开发视频软件。他常在个人博客（blog.miguelgrinberg.com）上撰写各类博文，内容主要涉及Web开发、机器人技术、摄影，偶尔也会有一些影评。他和妻子、四个孩子、两只狗和一只猫共同生活在俄勒冈州波特兰市。Twitter：@miguelgrinberg。

封面设计：Randy Comer　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/程序设计/Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-37399-1

ISBN 978-7-115-37399-1
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks